Title

Hardware Evaluation Analytical Modeling and Node Simulation: Benefits of Tighter GPU Integration

Permalink

https://escholarship.org/uc/item/8h57b16g

Authors

Austin, Brian
Bair, Ray
Barker, Kevin
et al.

Publication Date

2021-09-30

DOI

10.2172/1841724

Peer reviewed

ECP-HIHE01-35

Hardware Evaluation Analytical Modeling and Node Simulation: Benefits of Tighter GPU Integration

WBS 2.4.2, Milestone HI-HE-WG-3200

Brian Austin[3], Ray Bair[1], Kevin J. Barker[5], Anthony Cabrera[4], Andrew A. Chien[1,7], Nan Ding[3], Jesun Firoz[5], Khaled Ibrahim[3], Joseph Manzano[5], Vitali Morozov[1], Tan Nguyen[3], Leonid Oliker[3], Joshua Suetterlein[5], Li Tang[2], Jeffrey Vetter[4], Samuel Williams[3], Kazutomo Yoshii[1], Aaron Young[4]

[1]Argonne National Lab [2]Los Alamos National Lab,
[3]Lawrence Berkeley National Lab, [4]Oak Ridge National Lab,
[5]Pacific Northwest National Lab, [7]University of Chicago

September 30, 2021

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ECP-HIHE01-35

ECP Milestone Report

Hardware Evaluation Analytical Modeling and Node Simulation: Benefits of Tighter GPU Integration

WBS 2.4.2, Milestone HI-HE-WG-3200

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

September 30, 2021

ECP Milestone Report

Hardware Evaluation Analytical Modeling and Node Simulation: Benefits of Tighter GPU Integration

WBS 2.4.2, Milestone HI-HE-WG-3200

APPROVALS

Submitted by:

Samuel Williams HI-HE-WG-3200	Date

Concurrence:

Scott Pakin ECP Hardware Evaluation	Date

1. INTRODUCTION

In this report, we examine several emerging technologies of interest to the Department of Energy and its computational centers. These include: 1) quantifying the benefit of tighter CPU-GPU integration, 2) quantifying the appropriate CPU core:GPU ratio, 3) quantifying the penalty for CPU-GPU disaggregation, 4) quantifying the benefits of tighter GPU-GPU integration, 5) quantifying the benefits of unified memory, and 6) quantifying the benefits of tighter FPGA-GPU integration.

This report is structured as follows. First, we define the systems used for baseline comparisons and profile-driven model parameterization. We then provide background on the 16 HPC applications examined in this report. Third, we describe the methodologies used to quantify the benefits and penalties of the aforementioned technologies on our application suite. Note, each application-technology combination used the modeling and simulation methodology appropriate for that particular combination. Sections 5 through 10 provide detailed results and analysis of our modeling and simulation efforts on each technology for each application. Although Section 11 summarizes our results, we provide a few highlights here:

- Order of magnitude improvements in CPU-GPU integration will likely yield only a 30% increase in application-level performance for most applications (outliers approaching 2×) but will incur no substantive application changes.
- The majority of applications required only one or two Skylake-equivalent CPU cores per V100-equivalent GPU thereby facilitating single-socket integrated solutions.
- Fine-grained offloading of computation on a kernel-by-kernel basis to disaggregated accelerators (GPUs, FPGAs) will likely to incur at least a 2.5× slowdown. Any disaggregation solution must embrace coarse-grained offloading and technologies that incur far less than an order of magnitude impact on bandwidth and kernel launch overheads.
- Integrating more GPUs on a node is likely to improve performance by roughly 20% without program changes assuming constant PCIe bandwidth per GPU.
- CXL- or equivalent solutions allow for accelerator-accelerator pipelining of computation.

2. TEST PLATFORMS

In this report, we use a variety of techniques to understand how advances in technology can improve GPU-accelerated performance. For consistency and practicality, we make all of our performance estimates relative to a PCIe-attached NVIDIA V100 GPU but acknowledge that the same methodologies and technologies could be applied to other architectures. To that end, where profile or baseline data was needed, we used the Cori-GPU testbed at NERSC or the Summit supercomputer at OLCF. Experiments designed to ascertain the appropriate CPU:GPU ratio are normalized to Skylake-equivalent CPUs and V100-equivalent GPUs unless noted.

2.1 NVIDIA V100 GPU

NVIDIA released its V100 ('Volta') line of GPUs in 2018. Each Volta SM has 80 streaming multiprocessors (SMs), that provide a total of 7.8 TF/s peak double-precision performance. A Volta GPU has four stacks of HBM-2 high bandwidth memory, collectively providing 16 GB capacity at a peak bandwidth of 900 GB/s (Read+Write). Six NVLink links offer high bandwidth connections (25 GB/s/dir per link) to other NVLink-capable devices (e.g. other Volta GPUs or IBM Power9 processors) on the same node. One PCIe-3.0 x16 link enables a bi-directional 16 GB/s/dir connection to non-NVLink host processors.

2.2 CORI-GPU

Cori-GPU is the GPU-testbed partition of the NERSC's Cori system. It shares much of Cori's infrastructure, but unlike the rest of Cori, which uses the Cray Aries interconnect, Cori-GPU operates on a separate Infiniband subnet. Each of the 18 Cori-GPU nodes contains: a) 2 Intel Xeon Gold 6148 sockets, each with 20 'Skylake' cores operating at 2.40 GHz b) 8 NVIDIA Tesla V100 ('Volta') GPUs, c) 384 GB DDR4 memory, d) 1 TB on-node NVMe storage, and e) 4 Mellanox MT27800 (ConnectX-5) EDR InfiniBand NICs.

The topology of these components is illustrated in Figure 1. The purple lines represent PCIe-3.0 connections, each providing a peak bi-directional bandwidth of 16 GB/s per direction. Green and grey lines indicate NVLINK connections between GPUs. (These NVLINK lines are equivalent; different colors are used to help distinguish parallel lines.) Each NVLINK arrow represents a bi-directional bandwidth of 25 GB/s per direction.

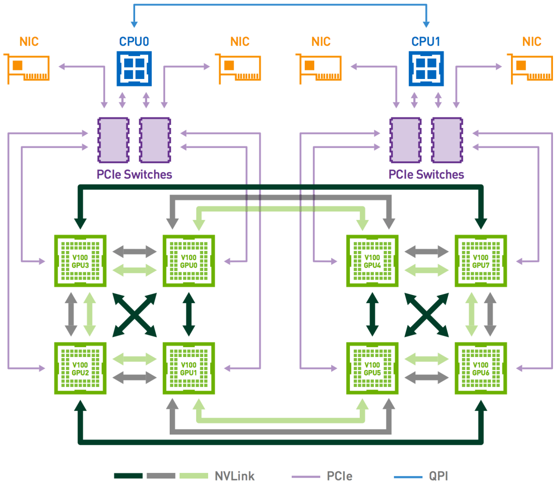


Figure 1: Cori-GPU node architecture. Figure adapted from NVIDIA: https://images.nvidia.com/content/pdf/dgx1-v100-system-architecture-whitepaper.pdf

2.3 SUMMIT

The Summit supercomputer was developed by IBM for use at OLCF. Summit consists of 4,608 compute nodes connected through a Mellanox EDR 100G InfiniBand, Non-blocking Fat Tree interconnect. Each Summit compute node contains: a) 2 IBM POWER9 Sockets, each with 20 P9 cores operating at 3.07 GHz, b) 6 NVIDIA Tesla V100 ('Volta') GPUs c) 512 GB DDR4 memory, d) 1.6 TB on-node NVMe storage, and e) 1 Mellanox InfiniBand ConnectX-5 EDR Infiniband NIC. The topology of these components is illustrated in Figure 2. On this platform (but not the Cori-GPU system described below) Address Translation Services (ATS) allow the GPUs to access the CPU's page tables directly.

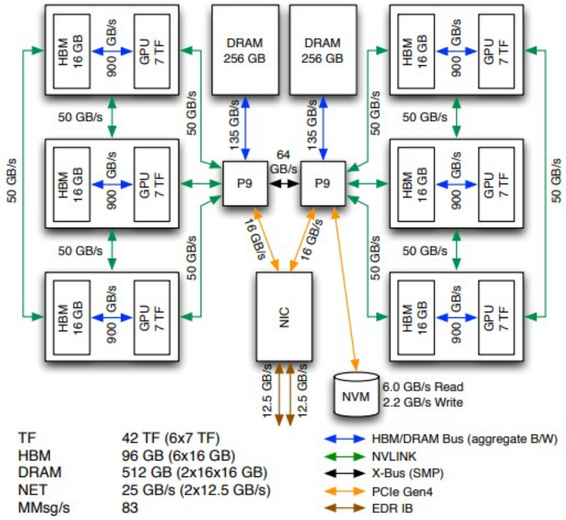


Figure 2: Summit node architecture. HBM and DRAM speeds are aggregate (Read+Write). All other speeds (X-BUS, NVLink, PCIe, IB) are bi-directional. Figure adapted from Summit System Overview: https://www.olcf.ornl.gov/wp-content/uploads/2019/05/Summit_System_Overview_20190520.pdf

2.4 NEWELL

The Newell system is installed at Pacific Northwest National Laboratory. It is designed to be a smaller facsimile of a Summit node with similar memory (16 GiB per accelerator and 1.1 TB of host memory), host (Power9 3.8 GHz with 32 cores, each with 4 threads) and network-on-chip (NVLink 2.0) fabric but with fewer V100 GPUs, from 6 to 4 connected in groups of two. It has four of these nodes wired through an InfiniBand Network.

2.5 GPU+FPGA SYSTEM

The experiments detailed in Section 4.8.2 were evaluated using a Linux Kernel-based Virtual Machine (KVM) configured on a server in the Experimental Computing Laboratory (ExCL) at Oak Ridge National Laboratory. The virtual machine is hosted on an Atipa server with dual-socket Intel Xeon Gold 6130 CPUs, 192 GiB of RAM, an NVIDIA P100 GPU, and a Xilinx Alveo U250 was used. The virtual machine was configured with 24 vCPUs, 92 GiB of RAM, and with PCIe passthrough to access the P100 and U250.

The NVIDIA P100 uses the Pascal microarchitecture, which houses 60 streaming multiprocessors (SM), with 64 CUDA cores per SM, and 16 GiB of off-chip, high-bandwidth memory. The Xilinx Alveo U250 is a data center acceleration PCIe card, which includes an XCU250 FPGA of the Xilinx UltraScale+ architecture, along with 64 GiB of off-chip, DDR4 memory. Both accelerators leverage a PCIe Gen 3x16 interface to the host.

3. HPC APPLICATIONS

In order to evaluate the potential performance gains from various forms of CPU, memory, and GPU integration and disaggregation, we must define an appropriate set of benchmarks. To that end, we explore performance on 16 proxy applications of interest to ECP, SciDAC, and DOE compute centers. These include: AMReX (Castro, WarpX, MFiX, and PeleC), BerkeleyGW (sigma and epsilon modules), GTC-P, SpTRSV, SW4Lite, MiniVite, Laghos, QuickSilver, HACC, LAMMPS, OpenMC, and an ORNL proxy application. We describe each of them below.

3.1 AMREX

Adaptive mesh refinement (AMR) is a powerful technique when the required computing resolution significantly varies across the simulated domain. Instead of having a uniform mesh with very high resolution, one can start with a coarse grid and patch a few sub-domains with a finer mesh. These patches form a second mesh refinement level, and the process can go on until we meet the resolution needed for the solver. It can be challenging to offload some AMR workloads on the GPU due to the complexity of the AMR data structure. For this milestone, we use a set of 4 applications from a block-structured AMR software framework called AMReX: Castro, WarpX, MFiX, and PeleC. Castro is an application suite for stellar and nuclear astrophysics problems. It includes a rich set of simulation codes such as hydrodynamics, nuclear reaction networks, and Poisson gravity. Due to such diversity, it can be expected that Castro may exhibit different performance behaviors depending on the simulated physics. WarpX models particle beam-driven and laser-driven plasma accelerators using the Particle-In-Cell method. AMR helps reduce the computation demand from WarpX's high-fidelity simulations over a large range of space and time scales. MFix is developed for multiphase flow reactor modelings at various scales. It employs a hybrid method of CFD (computational fluid dynamics) and DEM (discrete element method). PeleC models compressible and low Mach number combustion. PeleC uses Direct Numerical Solver (DNS) to simulate turbulent-chemistry reactions in conditions that are similar to that on real combustion devices. PeleC also uses embedded boundary to simulate complex geometries.

3.2 BERKELEYGW

The BerkeleyGW code computes excited state properties of materials using the GW formulation of many-body perturbation theory [7], and was a finalist for the 2020 Gordon Bell Prize. Of the several modules in the BerkeleyGW package, our profiling efforts have focused on the Epsilon and Sigma modules. Epsilon uses the random phase approximation to compute the polarizability and inverse dieletric matrices from a mean-field reference. Sigma uses output from Epsilon to compute the self-energy corrections to the mean-field eigenenergies. The material being simulated has a unit cell of 214 silicon atoms with a divacancy defect. (See [1] for details)

3.3 GTCP

The GTCP code uses a particle-in-cell (PIC) method to simulate plasma fusion solving the Vlasov-Poisson equation. The code is written in the C language using MPI for distributed memory, OpenMP for Multicore threading, and CUDA for offloading to NVIDIA GPUs. The code supports multiple levels of decompositions, including inter-node domain and particle decompositions, intra-node shared memory partitioning. The simulation progresses in a cycle of the following computational phases: charge deposition, particle push, particle shift, Poisson solve, etc. The most expensive routines typically involve processing the particle arrays. GTCP was a NERSC-9 procurement benchmark.

3.4 SPTRSV

The sparse triangular solve (SpTRSV) is often used to affect preconditioning in iterative linear solvers. SpTRSV is also used in direct methods and in least squares problems. As a result, SpTRSV is a critical computational kernel in many scientific applications. It solves a linear system of equations $Lx = b$, where L is a lower triangular sparse matrix and b is a dense vector. In recent years, substantial efforts have focused on single GPU SpTRSV [17, 14]. However, with more scientific insights derived from computation, the demand for ever finer-resolution problems calls for SpTRSV to exploit ever larger scales of parallelism. Unfortunately, given the slow pace in

HBM memory capacity scaling, one cannot guarantee the problem can always fit into a single GPU's memory. The low arithmetic intensity, complex data dependencies, and high GPU-GPU communication present immense performance impediments for multi-GPU SpTRSV. Thus, designing an efficient and scalable SpTRSV on modern multi-GPU HPC systems is imperative but challenging. During the design cycle, it is important to understand the performance bounds in terms of architecture constraints for SpTRSV.

3.5 LAGHOS

Laghos [3] is an application designed to solve the time-dependent Euler Equation of compressible gas dynamics in a moving Lagrangain frame using a high order finite element spatial discretization and explicit high order time stepping. This application encodes the structure of comprossible shock hyrdrocodes, e.g., BLAST code from LLNL. As one of its main components, Laghos uses the MFEM general discretization library. Laghos supports 2D / 3D unstructured meshes with both quadrilateral and hexahedral elements. Another feature of Laghos is its support for Runga-Kutta ODE solvers of orders 1,2,3,4, and 6. To solve these ODE systems, Laghos can use full assembly or partial assembly methods. Full assembly creates global mass and force matrices which are stored in a compressed sparse row (CSR) format. On the other hand, partial assembly methods only takes in consideration the local action of these matrices. These matrices are then used to perform the necessary operations. Due to the local action using the tensor structure of the finite element space, the memory footprint and behavior, and computational requirements are lower than the full assembly. Laghos uses MPI and OpenMP for their parallelism (both using MPI-based domain decomposition and OpenMP thread based). Additionally, CUDA and RAJA versions of the Laghos application are also available to support executions on the accelerators.

3.6 SW4LITE

SW4Lite [22] is the proxy application for Seismic Waves, 4th order application (a.k.a. SW4). This application is a seismic wave propagation code that solves the seismic wave equatios in displacement formulation using a node-based finite difference method. SW4 is in the EQSIM ECP project which aims to create simulation tools to predict earthquakes frequencies that might be pertinent to critical infrastructures (0-10Hz). Under EQSIM, a stand-alone SW4Lite proxy application was developed. This application solves the elastic wave equation with limited seismic modeling capabilities. SW4Lite is being used to explore several optimization opportunities for its parent application such as memory layouts, different concurrency models such as Hybrid MPI/OpenMPI and accelerated based models.

We use the gaussianHill-rev.in input for our experiments. This dataset is recommended by the application developers since it is a scaled-down problem that it is representative of an exascale challenge problem. Moreover, it exercises the relevant parts of the code while producing a similar performance. This input uses an analytical topology read from a file, and exercises the curvilinear kernels i.e., the most computational demanding kernels in the exascale problem.

3.7 QUICKSILVER

QuickSilver [23] is the proxy application from the Mercury workload (next generation general purpose radiation transport at LLNL) used to solve a simplified dynamic Monte Carlo particle transport problem. In this proxy app, particles interact with matter by collision and fission, and workloads with different reactions might exhibit distinct computational needs. QuickSilver matches Mercury's memory and network communication patterns plus its computational flow (i.e., branches). It is implemented in OpenMP, MPI and CUDA.

3.8 MINIVITE

The miniVite proxy application [9] implements a community detection algorithm named the Louvain method. It is an iterative method in which at each iteration, every vertex is checked to see if it is beneficial to move it to a neighboring community. The proxy application approximates the main application (Vite) by performing several iterations rather than converging. This mini application is important for our analysis since it is input dependent and irregular in access pattern. For our experiments, we use two distinct implementations that are optimized for CPU and GPU respectively.

3.9 HACC

HACC (Hardware/Hybrid Accelerated Cosmology Code) is an extreme-scale cosmological simulation code that runs on all available supercomputing platforms at very high-performance levels (Gordon Bell Award Finalist 2012, 2013). HACC uses a hybrid algorithm in its gravity solver, with the short-range computation being tuned to the system architecture. Gas dynamics in HACC is treated using CRK-SPH (Conservative Reproducing Kernel Smoothed Particle Hydrodynamics), a higher-order SPH scheme that does not suffer from difficulties in dealing with mixing and fluid instabilities. A number of subgrid models for gas cooling/heating, star formation, and astrophysical feedback mechanisms are part of the code. Current code development is led by an Argonne team and is supported by DOE's Exascale Computing Project.

3.10 LAMMPS

LAMMPS stands for Large-scale Atomic/Molecular Massively Parallel Simulator. This is a classical molecular dynamics simulation code with a focus on materials modeling. LAMMPS can model ensembles of particles in a liquid, solid, or gaseous state. It can handle atomic, polymeric, biological, solid-state (metals, ceramics, oxides), granular, coarse-grained, or macroscopic systems using a variety of interatomic potentials (force fields) and boundary conditions. It can model 2D or 3D systems with only a few particles up to millions or billions. LAMMPS is designed to be easy to modify or extend with new capabilities, such as new force fields, atom types, boundary conditions, or diagnostics. In the most general sense, LAMMPS integrates Newton's equations of motion for a collection of interacting particles. A single particle can be an atom or molecule or electron, a coarse-grained cluster of atoms, or a mesoscopic or macroscopic clump of material. The interaction models that LAMMPS includes are mostly short-range in nature; some long-range models are included as well. LAMMPS can be built and run on a laptop or desktop machine but is designed for parallel computers. It will run on any parallel machine that supports the MPI message-passing library. This includes shared-memory boxes and distributed-memory clusters, and supercomputers.

3.11 OPENMC

OpenMC is an open-source Monte Carlo neutron and photon transport code, which can run in parallel using a hybrid MPI and OpenMP programming model and has been extensively tested on leadership-class supercomputers. Monte Carlo methods of reactor simulation have a prohibitively long solve time on current-generation supercomputers. However, the embarrassingly parallel nature of the MC particle transport algorithm suggests that it should be an exceptional candidate for good performance scaling on exascale class supercomputers. Thus, exascale supercomputers offer the possibility of completing a robust, full-core nuclear reactor simulation with hundreds of nuclides and millions of geometric regions within a reasonable wall time, opening new avenues in reactor design. XSBench, a proxy application of OpenMC, abstracts the critical performance aspects of full-scale MC transport codes into a smaller package that is easier to port, run, and analyze on various novel and experimental architectures. XSBench executes only macroscopic neutron cross-section lookups, a critical computational kernel in MC transport applications that constitute 85% of the total run time of OpenMC. XSBench has been shown to mimic the computational requirements of full-scale MC transport applications accurately, so performance analysis done with XSBench will translate well to full-scale applications.

3.12 ORNLAPP

The application ORNLapp is a test application that was developed to represent a generic dataflow problem where outputs of a field-programmable gate array (FPGA) can be consumed by a GPU, or vice versa. This test application estimates the value of Pi using numerical integration of decompressed data by approximating

$$\int_0^1 \frac{4.0}{1+x^2}\,dx = \pi \tag{1}$$

as a sum of rectangles using the formulation

$$\sum_{i=0}^{N} \frac{4.0}{1+x_i^2}\Delta x \approx \pi \tag{2}$$

where N is the number of steps between 0 and 1, $\Delta x = \frac{1}{N}$, and $x_i = (i + 0.5)\Delta x$

The FPGA is responsible for decompressing a file that inflates to 1.6GB of double-precision values that will serve as input to the GPU to perform the numerical integration computation. The FPGA is using a Xilinx-provided implementation of `gzip`. While it is possible to compress and decompress data on a GPU, the parallelization schemes employed to accelerate this type of computation often are not amenable to GPU architectures, i.e., computation that is parallelizable across SIMD vector engines. However, the numerical computation in Equation (2) is well suited to be parallelized on GPU architectures, as none of the terms in the sum are dependent on previously computed iterations. This type of application is beneficial to study because it is indicative of future types of workloads on heterogeneous systems: FPGAs are shown to be very performant for sequential but pipeline-able computations, while GPUs excel when computations are regular with respect to branching and memory striding. While different types of computation are suited for differing accelerators, current practice requires the use of host memory in order to share data between the FPGA and GPU, since the accelerator memories are separate. Specifically, an explicit memory copy must be performed from the FPGA to the CPU, and additionally from the CPU to the GPU, in order to share data between accelerators. Additionally, since the decompression and integration happen in order, the computation can be pipelined to overlap communication and computation with both accelerators. Thus, our evaluation is centered around describing a more tightly integrated GPU-FPGA system to exploit the pipeline parallelism that exists within this and similarly structured applications.

The implementation of the decompression hardware and software are extensions of the Xilinx Vitis Libraries [30], which we will refer to as XVL. The XVL provides Xilinx C/C++ HLS implementations for a variety of different applications to make accelerating applications with FPGAs easier. The XVL uses the OpenCL 2.0 API specification to coordinate the host and FPGA. In this work, we use the implementations of the `gzip` decompression hardware found in the data compression sub-library of the XVL. Each XVL sub-library has 3 levels of abstraction–L1, L2, and L3–for reasoning about how to accelerate an application. L1 and L3 represent the lowest and highest levels of abstraction, respectively. In this work, we operate at the L2 level. L1 is the module level, and it provides hardware primitives for the components of the `gzip` decompression pipeline. These include Xilinx C/C++ HLS implementations of Huffman and Lempel-Ziv decoders, as well as modules that turn data from the FPGAs off-chip memory into AXI-streams and vice versa. The L2 level combines these data decompression primitives into pipelined kernels. In the ORNLapp, we use the hardware implementation of L2 `gzip` with no modifications. On the software side, we extend the classes of the L2 `gzip` application to make the data decompressed by the FPGA available via a host-side buffer that will be copied to the GPU.

The GPU implementation for the numerical integration of Pi is provided by researchers at the University of Bristol High Performance Computing Group [2]. This implementation leverages the OpenMP 4.0 specification for offloading computation to accelerators. GPU offloading of this computation is achieved through using the `target`, `map`, `teams`, and `distribute` constructs as defined in the OpenMP 4.0 specification.

Interoperability of the OpenCL and OpenMP programming models is required in order to coordinate the accelerators on the host side. In this application, mixing the two programming models is a non-issue because of the coarse-grained interaction between the FPGA and GPU. Specifically, the decompression task of the FPGA finishes entirely, the resulting decompressed data on the host is written to the GPU's memory via the `map` construct, and then the numerical integration on the GPU finishes in its entirety. More fine-grained interaction between FPGA and GPU and how to program such interaction in a CXL enabled system is discussed here and its implementation is a subject of future work.

4. METHODOLOGIES

No single modeling or simulation technique is appropriate for all applications or technology topics. As such, we employ a number of techniques including first-principals modeling and simulation, profile-driven modeling, binary instrumentation, etc... In this section, we describe each of these the techniques used in this report. Each of these techniques will be applied to one or more of the topics explored in subsequent sections.

4.1 PERFORMANCE MODELS FOR DIRECTED ACYCLIC GRAPH (DAG) COMPUTATIONS

Some numerical methods have a relatively simple communication pattern, such as stencils, which adhere to the Bulk Synchronous Parallel (BSP) model. Inter-processor communications follow the discipline of strict barrier synchronization. As such, these applications lend themselves to simple performance analysis. Conversely, DAG computations have a more complex communication pattern. Point-to-point communications can happen at any time between any two processes (depending on the sparsity pattern and the process decomposition) with no strict barrier synchronization. Therefore, DAG computations demand more sophistication.

SpTRSV is a typical DAG computation as Figure 3 shows. The data dependency of SpTRSV can be precisely expressed by a DAG based on the matrix sparsity patterns [8]. This DAG can be further refined according to the process decomposition. In the refinement, if two connected nodes belong to different processes (GPUs), this edge represents the GPU-GPU communication. Otherwise, it indicates the intra-node dependency. Thus, the in-degree from inter-processes of a node is the number of messages it needs to receive from other GPUs, while the out-degree to inter-processes of a node is the number of messages it sends to other processes (GPUs). After refinement, we can estimate the run time of each GPU by accumulating the computation time and communication time. The computation is estimated by using the bytes derived from each node, and the architecture memory bandwidth. The communication time is estimated by using the message size derived from each node, and the architecture network bandwidth and latency. Ultimately, the final SpTRSV time equals the accumulated run time of each DAG node on the critical path.

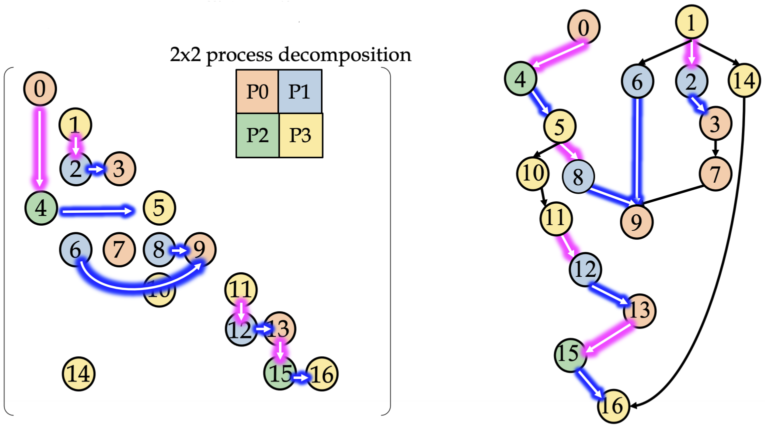


Figure 3: Recast a L matrix to a DAG. Blue arrows represent row reduction communication and pink arrows refer to column broadcast communication.

To estimate the run time of each DAG node, we need to perform architecture characterizations for memory bandwidth and network performance in order to parameterize our models. Let's take NVIDIA V100 GPU as an example, in the model, the memory bandwidth scales up to the peak 828GB/s [32] with the number of active super nodes in the system. The network bandwidths are parameterized by benchmarked message sizes using a round-trip ping pong benchmark via NVSHMEM [20]. We use all threads in thread blocks to put data to the target GPU (process) by `nvshmem_double_put_nbi_block`, and then perform a `nvshmem_fence`. Finally, we use thread 0 to send notification via `nvshmemx_int_signal`. When estimating the communication time, we perssimestically round the message size up to the next power of two to match the corresponding network bandwidth in the model.

Using this one performance model, we may evaluate CPU-GPU integration/disaggregation, CPU:GPU ratio, and GPU-GPU integration. A CPU-GPU integration can provide lower CPU-GPU latency and higher bandwidth which improve the NVSHMEM performance since NVSHMEM GPU-initiated communication performance is limited by the memory synchronization [10]. Therefore, the DAG model can quote the improved NVSHMEM performance to estimate the SpTRSV performance. The disaggregation penalty can be estimated by quantifying the sensitivity of NVSHMEM performance to PCIe bandwidth and latency. We can then use the adjusted NVSHMEM performance in DAG model to estimate the SpTRSV performance. The CPU:GPU ratio can be calculated by using the estimated performance that running more processes on one GPU. In that case, more processes equate to move of the DAG edges being on-node. Meanwhile, those processes equally shares the peak GPU memory bandwidth and flops. We can first update the communication type (inter-GPU or intra-GPU) between DAG nodes and then estimate the SpTRSV performance by changing the peak aggregate memory bandwidth of each process in the DAG model. For the GPU-GPU integration, we proxy with a super GPU with N times the FLOP/s, bandwidth, and memory but the same NVSHMEM bandwidth. We can then run N processes per super GPU. This is equivalent to moving the DAG edges on node but with each process achieving the same aggregated GPU memory bandwidth as the baseline. As such, we can also update the communication type (inter-GPU or intra-GPU) between DAG nodes and then use the adjusted super GPU memory bandwidth to estimate the SpTRSV performance.

4.2 PAREMETRIC PERFORMANCE MODELS FOR CPU-GPU INTEGRATION AND DISAGGREGATION

An integrated CPU-GPU architecture might reasonably be approximated by a pair (or cluster) of CPU and GPU processors whose performances are independent of each other; the primary affect of integration would be to improve the data- and control-flow *between* the processors. This could be acheived by improving the bandwidth or latency of the on-node-interconnect (e.g PCIe or NVLink), or incorporating of new accelerators to streamline specific inter-processor tasks such as kernel launch or GPU memory management.

A rudimentary peformance model that isolates these effects is given by

$$t_{total} = t_{CPU} + t_{GPU} + t_{MPI} + t_{CGdata} + t_{CGcontrol}, \tag{3}$$

where the total runtime (t_{total}) is decomposed into contributions from CPU computational kernels (t_{CPU}), GPU computational kernels (t_{GPU}), interprocess communication via MPI (t_{MPI}), CPU-GPU data transfer (t_{CGdata}), and CPU-GPU control-flow ($t_{CGcontrol}$). In this model, integration would impact only the last two terms. The simplicity of this model ignores the opportunity for overlap between contributions through asynchronous operations (e.g. kernel invocation, `CUDA_memcpy_async`, `MPI_Isend`).

The terms in Equation 3 can be determined using common profiling tools. The total runtime is readily obtained from the standard output output of the program being profiled. MPI time is measured with the IPM profiler.[26] Either of the NVIDIA profilers (NVprof [21] or nsight [19]) measure GPU and CPU-GPU activity: t_{CGdata} is the `[CUDA memcpy H2D]` and `[CUDA memcpy D2H]` calls, $t_{CGcontrol}$ is the sum of the non-memcpy cuda API calls and t_{GPU} is the sum of all other GPU activity. The CPU compute time was not measured directly, but assumed to be the difference between t_{total} and the other contributions.

Once these measurements are obtained, the runtime for a hypothetical integrated system can be estimated by scaling t_{CGdata} and $t_{CGcontrol}$ by hypothetical speedup factor, λ:

$$t_{integrated} = t_{CPU} + t_{GPU} + t_{MPI} + \frac{t_{CGdata}}{\lambda_{CGdata}} + \frac{t_{CGcontrol}}{\lambda_{CGcontrol}}. \tag{4}$$

Naive scaling is an obvious oversimplification that glosses over more complex effects; which could be included by more sophisticated models, but distinguishing those effects would require more elaborate profiling techniques. For the purpose of estimating the range of plausible speedups, a simple scaling approach is sufficient. However, one can often infer (*a posteriori*) from the average time per call whether the speedup factor for a particular function corresponds to latency or to throughput.

4.3 DETERMINATION OF OPTIMAL CPU-GPU RATIOS THROUGH ANALYSIS OF CPU STRONG-SCALING

Understanding the an application's optimal CPU:GPU ratio is often complicated by the hybrid use of MPI and OpenMP threads to exploit CPU parallelism. Each of these modes of parallelism might accelerate the same, different, or partly-overlapping phases of the run time. To orthogonalize these effects, we ran a series of jobs a constant number of GPUs, but different MPI and OpenMP concurrencies and fit their run time to an analytical model:

$$t_{total} - t_{mpi} = c_{seq} + \frac{c_{MPI}}{n_{MPI}} + \frac{c_{OMP}}{n_{OMP}} + \frac{c_{hybrid}}{n_{MPI} \times n_{OMP}} \tag{5}$$

In this model, n_{MPI} is the number of MPI processes *per GPU*, (this can exceed the number of GPUs if MPS or MIG is used). The number of OpenMP threads per MPI process is given by n_{OMP}. The c parameters have units of time and are determined by minimizing the RMS error of the model; c_{seq} is the contribution to the run time that is independent of CPU parallelism - it may include CPU operations that have no parallelism, time spent in CPU:GPU communication, or time spent computing on the GPU; c_{MPI} represents time spent computing on CPUs that is parallelized through MPI, but not OpenMP; c_{OMP} represents time spent computing on CPUs that is OpenMP parallel, but not MPI-parallel; c_{hybrid} represents time spent computing on CPUs that uses This model assumes that the CPU time does not overlap with GPU-time or MPI-communication time.

After the c parameters are obtained, the runtime for a hypothetical job can be modeled by

$$T_{model}(n_{MPI}, n_{OMP}) = t_{mpi} + c_{seq} + \frac{c_{MPI}}{n_{MPI}} + \frac{c_{OMP}}{n_{OMP}} + \frac{c_{hybrid}}{n_{MPI} \times n_{OMP}} \tag{6}$$

This job would require $n_{core}(n_{MPI}, n_{OMP}) = n_{MPI} \times n_{OMP}$ cores per GPU. In this idealized model, the asymptotic performance maximum occurs when n_{MPI} and n_{OMP} approach infinity; the corresponding runtime is $T_\infty = t_{MPI} + t_{OMP}$. We define the optimal ratio of CPU cores per GPU to be the smallest number that meets 90% of the asymptotic performance. This can be found by numerically minimizing n_{core} subject to the constraint $T_\infty / T_{model}(n_{MPI}, n_{OMP}) \geq 0.90$.

4.3.1 Optimal CPU-GPU ratio for AMReX applications

For AMReX applications, we use a simpler method to estimate the optimal CPU-GPU ratio. That is, we keep increasing the number of cores until the performance improvement is negligible. There are a couple of constraints though.

- AMReX applications don't exploit both CUDA and OpenMP (one or the other), forcing us to have multiple MPI processes sharing a single GPU.
- The number of MPI processes per node is limited to 8. A ratio higher than 8 will not be explored.
- The ratio on our testbed is 5:1 (40 CPU cores vs. 8 GPUs per node). Thus, a ratio of 8 if suggested will not be tested on the whole node since such an experiment would need 8×8=64 cores.

4.3.2 Optimal CPU-GPU ratio for applications with non-linear scaling

Some application that may not successfully oversubscribe the use of the GPU because of the persistent allocation of arrays. For such application, we can only vary the openmp number of threads. We model the behavior of changing the process count using a decaying function in the form of $T = \alpha + \beta * e^{-\lambda \times c}$. After fitting our data to find the optimal values of α, β, λ, we estimate the performance at large core count. We use the cutoff of 90% efficiency relative to ultimate performance to decide on the number of core per GPU, i.e., the c parameter.

4.4 A SIMPLE SIMULATION INFRASTRUCTURE (SSI)

One of our main objectives for this project is to evaluate the implication of future changes in hardware architectures on application performance. In particular, we are interested in assessing the effects of aggregation and dis-aggregation of CPUs and accelerators (i.e. to evaluate the trade-offs in closeness/distance of placement of CPUs and GPUs) on the performance of different applications, discussed earlier. For this purpose, we have developed a simple SimPy-based simulator in Python, Simple Simulation Infrastructure (SSI). Without resorting to detailed simulation of all hardware components in the system, SSI focuses on simulating parts of the system to assess whether changing the placements of different computational resources (CPU, GPU, etc.) and varying different link-specific parameter values would effect the performance of the applications that are of interest to the scientific community. This simulator can vary the bandwidth of the links, available memory of computational components and the number (width) of connections (links) between these components. This infrastructure is appealing because, with SimPy's simple and intuitive simulation components, we developed rapid prototypes of the simulated conditions based on queuing theory. Another advantage of our SimPy-based simulator is that it can simulate queue contention (based on link parameters connecting different computational resources) to a degree which fits our purpose to understand the effects of aggregation. Moreover, since links and queues can be parameterized, large design space exploration can be conducted quickly.

In this simplified simulation framework, a system consists of queues, links, and workers. A queue represents available memory attached to a computational resource in the system. An entry in a queue can be abstracted as a whole data structure (binary tree, linked list, an array or others) or a smallest memory quanta such as byte, bits, word, page, cache line, etc. A link represents a structure connecting different computational components. Links can be parameterized by propagation time and transmission delays of transferred bytes as well as the number of links. The computational resources (CPU, GPU) act as "workers" and retrieve"work" from the queues. Based on the available memory (simulated by the queue lengths), message transfer time over the links/connectors and ideal GPU kernel execution time, the workers simulate the actual execution time of an application. As we vary different parameter values (memory/queue size, link bandwidth, width etc.), the total execution time of an application can also change. We capture this predictive behaviour with our simulation framework. Initially, a kernel execution time on the GPU is collected with Nvidia's Nvprof profiler.

A typical run of the simulation framework requires two sets of inputs. The first one consists of application-specific profiles, including information about the important kernels, memory requirements and kernel launch and execution time. Additionally, a second set of parameters is also given as input to the SSI simulator that includes hardware parameters such as link bandwidth, computational costs, number of devices, queue sizes etc. Once the simulation completes, the framework outputs an estimated execution time and utilization. A graphical view of the simulation components and their interaction is presented in Figure 4.

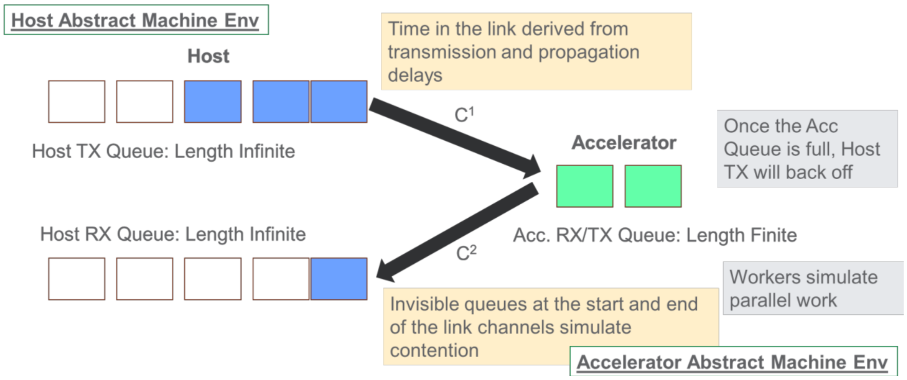
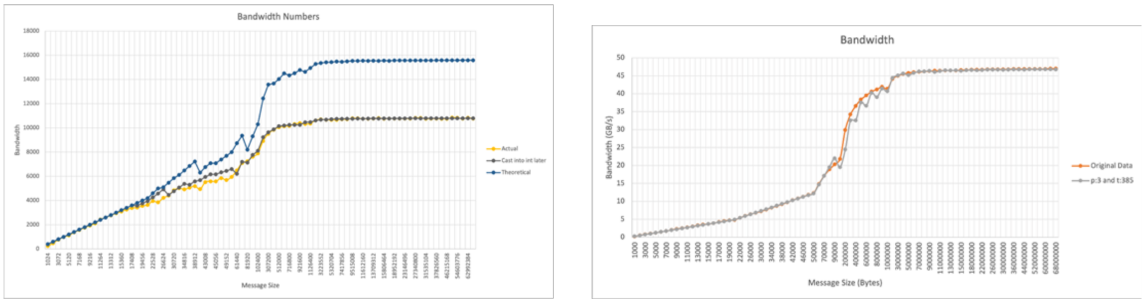

Host Abstract Machine Env
Host
Time in the link derived from transmission and propagation delays
C1
Accelerator
Once the Acc Queue is full, Host TX will back off
Host TX Queue: Length Infinite
Host RX Queue: Length Infinite
Acc. RX/TX Queue: Length Finite
C2
Workers simulate parallel work
Invisible queues at the start and end of the link channels simulate contention
Accelerator Abstract Machine Env

Figure 4: The Simulation Framework Overview

4.4.1 Link's Design Space Exploration

To ensure the correctness of our simulator, we check the simulated bandwidth against actual reported bandwidth of an integrated CPU-GPU system by varying the message sizes transferred over the links. Since we consider the links between CPUs and GPUs as one of the most interesting aspects of future systems that quantifies (dis)advantage of (dis-)aggregation, we consider PCIe-3 and NVLink as the hardware component for validating our simulation. The parameter values for these link technologies are obtained on a DGX-1 system with Volta GPUs by running the bandwidth test available in CUDA toolkit.

The results from the simulation and from actual run on the hardware are shown in Figure 5a and Figure 5b.



(a) PCIe 3 Results: Theoretical vs Real vs Sim

(b) NVLINK2 Results: Real versus Simulated

Figure 5: Bandwidth Results from real versus simulated runs

As can be seen from the figures, the bandwidth values reported by the simulator closely matches experimental results on the hardware and helps to provide confidence in the accuracy of simulated behavior of the link part of the simulation infrastructure.

4.4.2 Analytical Model for Workload Characterization

Besides characterizing and predicting the performance of the applications mentioned earlier on future architecture, SSI also incorporates an analytical model to enable simulating the performance of additional workloads. This analytical model, named Memory Divergent Model (MDM), was proposed in [29].

The model takes into consideration two important limitations of the previous analytical models. First, due to not considering the limited number of Miss Status Holding Registers (MSHR) in accelerators, most of the analytical models previously proposed erroneously overcompensate for memory-access latency hiding techniques on GPUs. Second, request for memory access by memory divergent applications incur heavy congestion on the Network-on-Chip (NoC) and the DRAM subsystems which can result in predicted instruction per cycle (IPC) with large error margin.

To factor in the effects of these two phenomena, MDM proposes the following analytical model, that includes the number of stall cycles resulting from the limited number of MSHR and NoC/DRAM congestion using Formula 7:

$$IPC_{SM} = W * \frac{\sum_{i=0}^{no.\_intervals} number\_of\_instructions}{\sum_{i=0}^{no.\_intervals} C_i + S_i^{MSHR} + S_i^{NoC} + S_i^{DRAM}} \tag{7}$$

where the S_i-s represent stall cycles encountered due to the limited number of MSHR registers, queuing in the NoC and DRAM subsystems. By varying different workload-specific, as well as hardware parameter values, MDM model can help us find the IPC for an application workflow. This is specially useful in the scenario where the user may be interested in experimenting with an application-agnostic (unknown) or slightly perturbed workload pattern as hardware parameters are changed. Afterwards, we can use the estimated IPC value and total instruction count to recalculate the clock cycles (execution time) of a different instruction stream. In our experiments, we assume that the IPC calculated with this formula proposed in MDM is representative of the application running at saturation level.

This assumption holds at least for Laghos as seen in Figures 6 and 1. To verify the predicted IPC value calculated by the MDM model, we also collect the IPC values for the same set of applications with NVidia's nvprof profiler.As can be observed from Table 1, the profiler-reported IPC values and the estimated IPC values by the MDM model is reasonably close. This validation provides us with enough confidence to apply MDM model for prediction, when a vendor-supplied profiler can't be used, due to various reasons (hypothetical but interesting workload pattern, overhead introduced by the profiler etc., for example).

We use the values of Table 1 to calculate the kernel time for our simulation and used the NVLINK and PCI-E traffic, and the kernel execution times to set up the simulation environment of the largest kernel of the respective workflow.

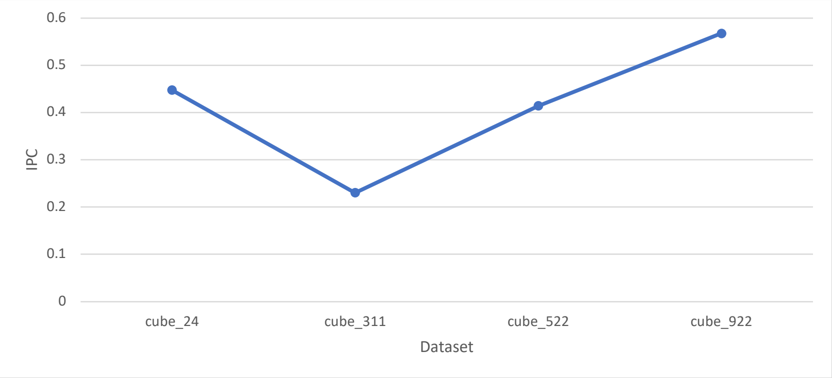


Figure 6: Laghos IPC across different Data sets

App	Dataset	Instruction Executed	IPC (orig)	IPC (calculated)
SW4Lite	guassinhil_rev2	740557990	0.9	0.33
MiniVite	RG scale 24	138628618	0.46	0.7
QuickSilver	50000 Particles	2000196509	0.314	0.38
Laghos	cube_24	1243562	0.497	0.6

Table 1: Applications Instruction streams, IPC calculated versus observed

4.5 ACCELERATOR CENTRIC COLLECTION OF COMMUNICATION PROFILES

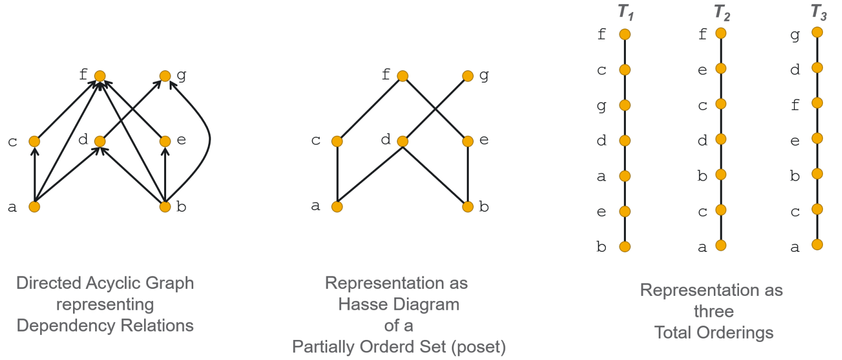
As scientific applications are becoming more complex, several efforts are ongoing to understand the implication of changes in the hardware design on their underlying behaviors. In this section, we ask the question: besides considering the (dis)advantage of CPU-GPU (dis)aggregation, is there a necessity to assess the performance penalty for GPU-GPU (dis)aggregation, in the context of ECP proxy apps? To answer this question, we develop a methodology and tool to find out how much opportunity for parallelism is available in these applications written as is (in terms of concurrently executed kernels), whether the scientific applications of interest leverage multiple GPUs to run concurrent kernels and whether the dispatched kernels on different GPUs communicate with each other over the *link*. In the past, the most time-consuming part of these application was the network (i.e. "scaling out" was challenging). However, as the HPC architectures are evolving towards a more accelerator-centric designs, the performance of the application is also decided by the communication between the accelerators and their hosts (i.e. "scaling up" within a single node is becoming challenging too). For these reasons, the understanding of the accelerator behavior and effects are very important for the application running on these architectures.

One approach to understand the interactions among hardware substrates and applications is to extract the communication profile of the application over several runs and construct a communication (composed of either MPI communication primitives or Host / Accelerator Calls) Directed Acyclic Graph(DAG), where each vertex in the DAG represents a kernel and a directed edge between two vertices represent dependencies between the kernels (i.e. result of the computation from one kernel is required for the dependant kernel to proceed).

Such a DAG can be used to understand the application behavior (for example, available concurrency can be depicted by sibling nodes in one level of the DAG, directed edges can capture dependencies among the kernels etc.) under different conditions (for example, by changing inputs, topologies etc.).

Such communication profiles can be extracted independently with an MPI profiler (e.g., PMPI) or a binary instrumentation framework for accelerators (e.g., NVBit). Based on these observations, we use the FENATE [33][1] (Fast Evaluation of Network Architectures – Toolchain and Environment) tool suite to extract these profiles to create communication DAGs for both accelerators and network topologies.

[1] which itself build upon PMPI and NVBit



Directed Acyclic Graph representing Dependency Relations

Representation as Hasse Diagram of a Partially Orderd Set (poset)

Representation as three Total Orderings

Figure 7: FENATE different internal formats

4.5.1 Extracting the DAG: the poger Component of the FENATE Framework

FENATE can be seen as a qualitative view of the communication networks. It is a trace based framework designed to extract communication DAGs and simulate their behaviors on different computational substrates. It can be considered as a zeroth order framework that helps current frameworks (like SST [16]) with their analysis. It is mostly written in Python and it accepts inputs from two tracing both developed at Pacific Northwest National Laboratory: one called the Visual Introspective engine for System Under Stress (VISUS) designed for OpenMP and MPI and another tool based on NVBit (called NVDAG) to capture the accelerator kernel behaviors.

FENATE is composed of two components: the Partial Order Generation from Experimental Results (poger) and the Lightweight Message Passing in Python (LiMPPy). The poger subcomponent takes different program traces and creates partial ordered sets for each run. Each of these posets are used to create cover relationships using transitive reductions to extract true dependencies. The internal structure of poger is presented in Figure 8

To extract the DAGs, the tracing tools are required to insert delays into the activity of interest (i.e. MPI sends/recieves or GPU kernels). These delays help to break false dependencies that arise by scheduling across the communication pairs. These delays can be a pattern or random accordingly to the computational substrate being analyzed. After this, we collect traces from across several runs. In this way, the probability of a false dependency passing the analysis is reduced. FENATE reads and coalesce them into other graph representations (i.e., Hasse diagrams) for further analysis. These different formats are showcased in Figure 7.

On the other hand, the LiMPPy subcomponent was designed as a lightweight simulation model to replay the computational graphs produced by poger. LiMPPy accepts generative functions to generate routing, tables, topologies and different link properties. The framework has been tested with ECP mini applications like Sweep3D, SW4Lite and MiniVite on simulated computational substrate such as SimFly, BundleFly, CLEX, LPS. Tofu, Mesh and Fat trees topologies.

For the purpose of this report, the poger subcomponent is of special interest since we have been developing the accelerator based poger part of this suite. We are using it to extract the computational DAGs from accelerator / GPU workflows using the NVBit framework.

4.5.2 NVDAG

The NVBit library [28] is an NVIDIA library for binary rewriting code for CUDA applications. It works in inserting trampolines at the beginning and the end of CUDA api calls and allows the collection of runtime and user defined information about the running kernels. To feed poger with CUDA based traces, we created a small NVBit tool called NVDAG. This tool inserts calls to save kernel id, running stream, and timestamps to create the partial order sets based on the CUDA kernels ordering across the GPU streams.

Based on the inherent difference between the MPI based and GPU based flows, the poger sub-component needs to restructure certain parts to accommodate these changes. Due to the global clock of the accelerator (the host clock), poger does not need to relabel and restructure the traces to keep the dependencies consistent. Since the only vehicle of concurrency are the streams in these examples, stream labels and certain degree of dynamic

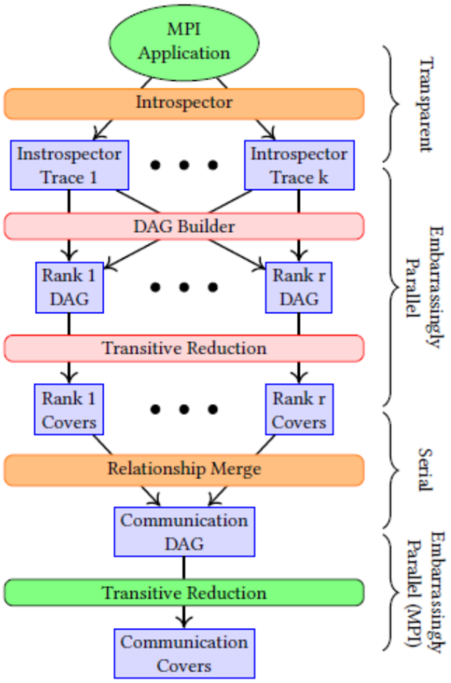


Figure 8: The poger Architecture

scheduling is required to fully extract the DAGs. Because of the nature of NVBit, the delay is not as flexible as in the other cases, so a random number is used for the delay.

We use the NVDAG framework to classify the concurrency available to kernels of our workflows. This analysis helps us to understand if the workflows are naturally growing to more GPUs or not. Moreover, this tool helps to capture synchronization behavior of other programming layers (such as MPI) which not be possible with CUDA centric tools.

Finally, while calculating the CPU:GPU ratio, we profile the workflow to get their utilization and then we conduct a set of over subscription experiments with several MPI ranks and a single GPU.

4.6 BYFL: COMPILER-BASED APPLICATION ANALYSIS

4.6.1 Byfl Instrumentation

Byfl is an LLVM based non-invasive instrumentation tool that developed at LANL and works as a compiler wrapper that supports instrumentation of programs written in C/C++/FORTRAN. Byfl could be viewed as a software performance counter that counts various operations (e.g., flops and load/store operations) of a specific run for an instrumented program by Byfl at the LLVM IR level, which is independent on hardware. As Byfl operates at the LLVM IR level, it counts various operations and collects dynamic execution information that are not available from hardware performance counters. For example, Byfl can profile the memory access patterns by detecting the stride widths of neighboring load/store operations. We use Byfl to instrument programs and focus on identifying both the regular and irregular memory (i.e., load/store) operations. Specifically, we measure the stride widths of all memory loads and stores and classify them into either regular or irregular memory operations by identifying whether their stride widths are within 128 Bytes (i.e., a relatively large cache-line size).

4.6.2 Byfl based Unified Memory Modeling

To model the performance of the unified memory programming model for CPU+GPU architectures, we extend the roofline performance model to include the considerations of data transfer and memory access patterns (i.e., regular and irregular). The total GPU execution time could be estimated by using the following equation:

$$T_{GPU} = \max(t_{flop} \times n_{flop},\ \ t_{reg} \times n_{reg} + t_{irreg} \times n_{irreg} + t_{h2d} \times n_{h2d} + t_{d2h} \times n_{d2h}) \tag{8}$$

In this equation, the numbers of regular and irregular memory operations (i.e., n_{reg} and n_{irreg}) are measured by using Byfl. The data transfer data sizes (i.e., n_{h2d} and n_{d2h}) are defined by users. The timing information (i.e., t_{xx}) are measured by using the STREAM and pointer-chasing memory benchmarks of evaluating GPU memory performance. With the application parameter values and the obtained hardware parameter values, we can use the extended roofline performance model to predict if a specific program can benefit from the unified memory programming model.

4.7 AGOCHART: COARSE-GRAINED PERFORMANCE MODELING

Preparation for Aurora and refinement of its design presents unique challenges because Intel plans to use discrete HPC GPUs, which are still under development in a top scientific computing system for the first time. No family of Intel discrete GPUs exists to use as reference models for node performance. This project's discussions with Argonne Leadership Computing Facility's Performance Engineering Group identified the need for a new kind of node modeling tool, one that could use information about the performance of existing leadership applications and project it to node configurations that did not yet exist. Hence, the concept for AgoChart was born.

AgoChart's objective is to enable early and rapid node performance estimation for ECP applications targeting the Aurora system. In addition, we expect the AgoChart tool to diagnose the balance of computing and data movement cost, tradeoffs in movement between CPU and GPU. Parameters allow simulation of different GPU generations, GPU specifications, and node configurations (e.g., GPU count). AgoChart is a coarse-grained performance modeling tool aimed at application developers and performance engineers. AgoChart's few parameters enable quick models to evaluate changes in CPU-GPU communications bandwidth and data transfer and kernel invocation latency, the number and relative performance of computation units within a GPU, and the number of GPUs on a node. These parameters enable projection from one generation of GPU node to the next, or some future, speculative GPU.

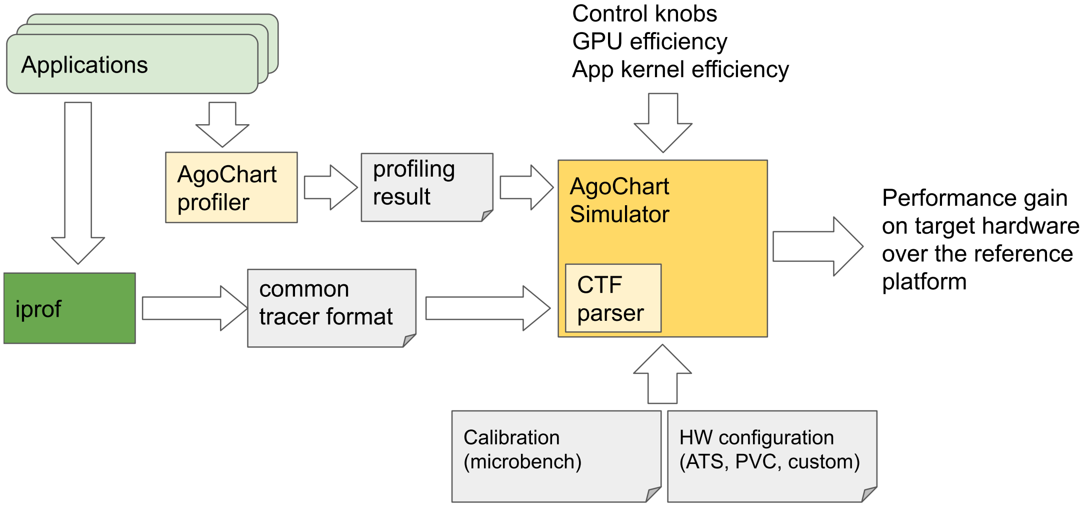


Figure 9: AgoChart simulator tool

AgoChart utilizes a trace of host-GPU requests recorded from an actual application on an existing CPU-GPU node (Figure 9). It records the data size of CPU-GPU data transfers and the execution times of each GPU task issued by the CPU. That information alone has value, as the time graph of communication and computation reveals communication bottlenecks. However, AgoChart can also take that time series and project it to the parameters of a new system. It will answer questions about how well an application will continue to run on a scaled-up GPU with more or faster computation units or with different ways/speeds of connecting GPUs and

CPUs. This insight helps application developers better understand where to put their effort in preparation for Aurora and other exascale systems.

Our current methodology requires a program that uses OpenCL as a backend, a native OpenCL program, or an OpenMP program that uses the OpenCL runtime. Support for oneAPI (data parallel C++) is under development. It can project for forthcoming Intel discrete GPUs with proprietary parameterizations. As for a tracing tool, we use the iprof tool, which is developed by Argonne ALCF or the AgoChart profiler too that captures trace data and timing for OpenCL API calls. The AgoChart simulator takes that trace data and creates a discrete event simulation for a different CPU plus discrete GPU combination. AgoChart can be calibrated by application teams (customized based on their knowledge) or generically based on hardware parameters such as the number of GPUs and estimates of kernel and GPU efficiency. The result is an event time trace and performance prediction based on the specific node configuration. As requested by ALCF, we examined the performance characteristics of OpenMC, and the LAMMPS and HACC application kernels.

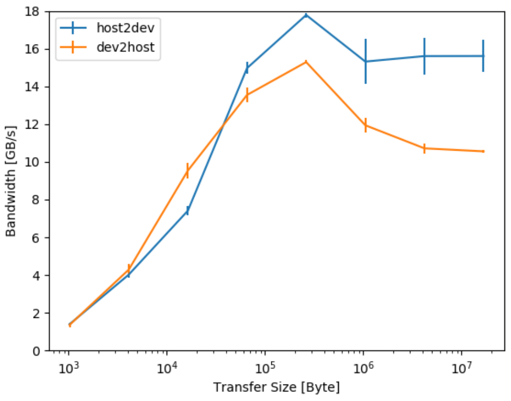


Figure 10: Host-device bandwidth: A100

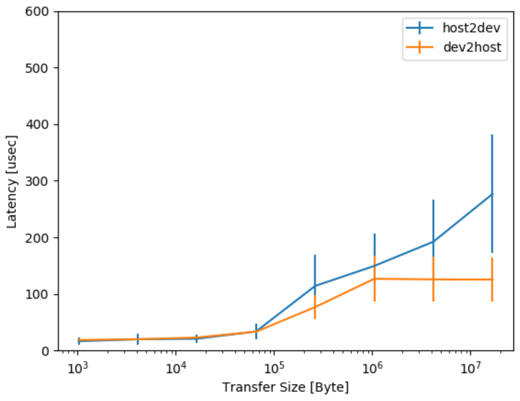


Figure 11: Host-device latency: A100

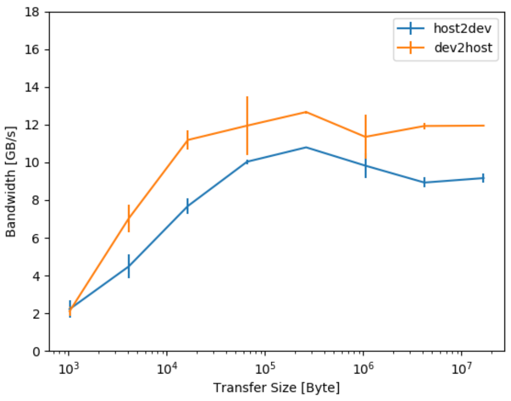


Figure 12: Host-device bandwidth: V100

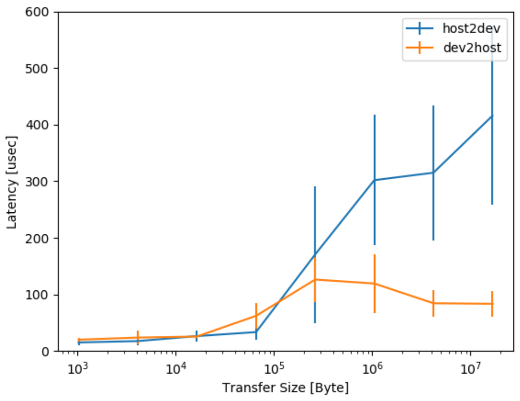


Figure 13: Host-device latency: V100

Intel discrete GPUs are still under NDA as of this writing. Only prototype GPUs are available internally at Argonne; we cannot publish any performance results we obtained directly from the prototype GPUs. To answer the questions in this milestone, such as the benefit of tighter CPU-GPU integration, we used measured performance from existing HPC-class discrete GPUs (NVidia A100 and V100) to calibrate our model, more specifically the invocation, data transfer latency, and bandwidth, using an OpenCL-based micro-benchmark we have developed that measures the bandwidth and latency of data transfer between host CPU and GPUs. Figure 11 and 13 show measured latency on data transfer between CPU and GPU for different transfer sizes. In addition, Figure 11 and 13 show measured bandwidth. The 'host2dev' label is associated with input data to GPU, and the 'dev2host' is associated with output data from GPU. Vertical lines on data points are the standard deviation. Timing is measured using OpenCL's event-based timer (e.g., clEnqueWriteBuffer for host2dev); bandwidth is the buffer size divided by the transfer time, and latency is the difference between the end time of the previous transfer event and the start time of the current transfer event. All OpenCL calls in the benchmark code are properly synchronized. This measurement includes PCIe transfer and moving from/to host main memory and data movement inside OS. Measured bandwidth and latency in existing literature [15, 13] show similar characteristics to our measurement.

4.8 GPU AND FPGA INTEGRATION STUDY AND CXL EVALUATION

While GPUs have been the de facto choice for hardware accelerators, FPGAs have also been gaining traction. One of the major advantages of FPGAs is that its reconfigurable fabric can be tailored to a specific application or domain. This allows developers to architect custom hardware logic for a given application without having to go through the rigorous and costly experience of designing an Application-Specific Integrated Circuit (ASIC). The use of FPGAs, however, should not be viewed as a replacement to GPUs as both GPUs and FPGAs have complementary strengths. In fact, both GPUs and FPGAs can be attached as PCIe cards within a single node and used together to accelerate an application. Outside of nascent research, though, there is no off-the-shelf infrastructure for easily integrating the custom compute capability of FPGAs with the wide SIMD functional units of a GPU.

In our work, we first created the ORNLapp discussed in Section 3.12 to demonstrate one way FPGAs and GPUs can be used to accelerate a single application by leveraging the strengths of both accelerators. This application also highlights how both FPGA and GPU programming has matured to be easier to use. The design of the FPGA logic is created using a High-Level Synthesis (HLS) workflow with a Xilinx Acceleration Library to perform decompression. The GPU logic leverages OpenMP 4.0-style offloading to utilize the GPU. Host and device communication is handled through OpenCL for the FPGA and through OpenMP for the GPU. This application highlights that the tools used to leverage the accelerators can be different for each accelerator type. This application also shows the basic interoperability capability of using OpenMP for GPUs and OpenCL for FPGAs.

Additionally, our work explores a new protocol called Compute Express Link (CXL) that can help address the issue of tighter integration between FPGAs and GPUs. The CXL protocol is an open interconnect standard that is implemented on top of the PCIe 5.0 specification. It provides low-latency and cache-coherent communication between host processors and devices such as accelerators, memory buffers, and smart I/O devices. Effectively, this protocol aims to add tighter integration between compute components within a node, which will enable a shared unified memory space with caches on each device and latency times similar to CPU cross-socket communication. Our goal in this work was to evaluate how the CXL protocol could provide tighter integration within a node that contains both a GPU and FPGA.

Our approach to evaluating GPU-FPGA integration and the CXL protocol was twofold. First, we established a baseline measurement using currently available hardware and tools. As a test application, we estimate the value of Pi using numerical integration. See Section 3.12 for more information about this test application. Second, we use a numerical CXL model to evaluate the expected speedup obtained from leveraging CXL to achieve tighter integration

4.8.1 CXL Background

CXL is a new open industry-standard interconnect, which promises to provide high-bandwidth, low-latency communication between the host processor and connected devices such as accelerators like GPUs and FPGAs as well as other devices like memory devices and smart I/O devices [24, 25, 5, 6]. CXL presents a cache-coherent, unified memory with a shared address space between the host processor and any of the attached devices. Local device and CPU caches are used to increase access speed for memory stored on a remote device. CXL is a non-symmetric protocol with the CXL caching agent running on the host processor to manage the state of cached memory at a 64-byte cache-line granularity. The unified memory can be made up of memory located on any or all of the CXL devices. CXL coherence bias allows a device to directly access memory stored on that device without needing to go through the host coherence agent when no other devices are requesting that page of memory. The two modes of CXL coherence bias are device bias, where devices have direct access to the memory and host bias, where all cache operations go through the coherence agent on the CPU.

CXL is built on top of the PCIe 5.0 physical layer infrastructure and the PCIe alternate protocol. CXL supported devices are backward compatible to PCIe through Flex Bus. Flex Bus link training starts at PCIe Gen1, then negotiates the link protocol up to the CXL alternate PCIe protocol, (or the best mutually supported link protocol).

The CXL transaction layer is comprised of three sub-protocols which are dynamically multiplexed on a single physical link. These three protocols provide support for I/O (CXL.io), caching (CXL.cache), and memory (CXL.memory). Only the CXL.io protocol is required to be implemented in a device, with different classes of devices implementing different sub-protocols. CXL.io has the same functionality as the existing PCIe protocol and enables device discovery, configuration, initialization, register access, and direct memory access (DMA) without the unified memory space. CXL.cache provides devices with access to the unified cache coherent memory managed by the host. This protocol is required to give devices access to the unified memory space. CXL.mem allows devices to provide memory to the unified cache coherent memory managed by the host. This protocol allows the host to access the memory attached to the device so it can be included in the unified memory space. Both the CXL.cache and CXL.mem stacks are optimized for low-latency transactions. They use fixed message framing and a separate transaction and link layer from CXL.io. CXL.io uses a stack that is largely identical to a standard PCIe stack.

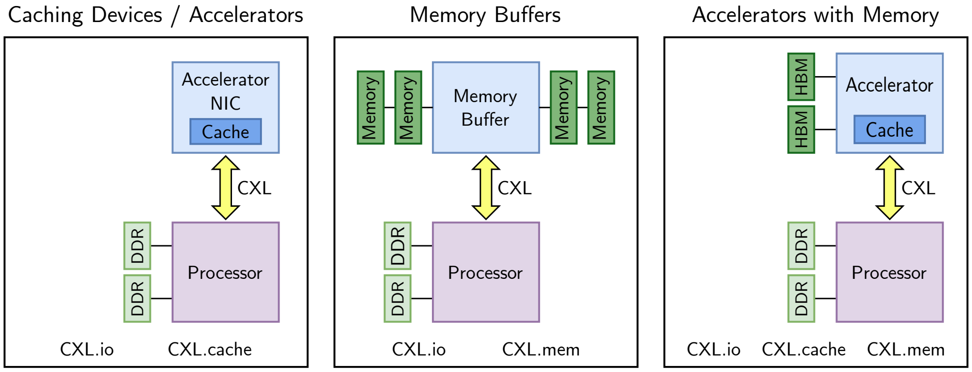


Figure 14: CXL devices classes [6].

As mentioned above and shown in Figure 14, different device classes implement different subsets of the CXL protocols. A caching device/accelerator like a SmartNIC would implement the CXL.io and CXL.cache protocols. These devices would have access to the unified cache coherent memory and would have a local device cache. This class of devices access the unified memory, but don't have large memories of their own. The next type of device is a memory buffer, which implements the CXL.io and CXL.mem protocols. Devices in this class include memory expansion cards. These devices provide memory for the unified memory space, but do not use the memory. CXL memory devices like this fill the gap between DRAM and performance SSD and provide an easy interface to connect new emergent memory technologies. The final class of devices, accelerators with memory, is the category that includes GPU and FPGA accelerators. This class uses all three protocols, CXL.io, CXL.cache, and CXL.memory and both provide local HBM memory to the unified memory space and access the unified space through a cache.

The CXL device class is for acclerators with memory, and is the class this paper focuses on since it includes FPGA and GPU devices. In this paper, we assume that the accelerator is operating in host bias mode since we assume that there is tight sharing of the data between different devices. If the data sharing is done more coarsely, then device bias mode can be used for higher memory performance. To use device bias mode, more thought will be necessary to determine which device will store the data being operated on. Although each device will be able to access all the memory via the unified address space; devices will additionally be able to access the address ranges stored in local memory directly through device bias.

4.8.2 Measuring Performance for GPU and FPGA

In order to collect performance measurements for the GPU and FPGA, we use the native profilers of the respective accelerator. For the GPU, we use a combination of the command line utility `nvprof` and the NVidia

Visual Profiler `nvvp`. These profiling tools allow us to capture the amount of time spent in various parts of the GPU portion of the application. Specifically, we are able to record the amount of time spent transferring the decompressed data from host to device (which is tagged as `CUDA memcpy HtoD`, and the amount of time elapsed while performing the numerical integration (which is tagged with the symbol name given to the auto-generated GPU kernel, `__omp_offloading_3b_24c1421_main_l85`.

On the FPGA side, we use the run summaries and logs generated by the Xilinx Runtime Library (XRT) [31] to capture the amount of time spent on the FPGA, and its interactions with the host. We visualize this data with the Vitis Analyzer tool provided by Xilinx. The FPGA decompression pipeline is executed such that it overlaps data transfer to and from the host with kernel execution, in order to more efficiently decompress the data. The sizes of these transfers are fixed at runtime but are statically parameterizable on the host side at compile time. The host read transaction is always 2× larger than the write transfer. We are interested in the durations and bandwidths of each of these chunked transfers at different sizes. To compute this, we run the whole application 100 times using 7 different configurations. The space of configurations is as follows:

$$\text{Configuration} \in \{8{-}16, 16{-}32, 32{-}64, 64{-}128, 128{-}256, 256{-}512, 512{-}1024\}$$

where each element in the set contains the size of the write and read transactions, respectively, for a given configuration, separated by a dash. All elements of the set are measured in mebibytes (MiB). For each configuration, we average the read and write durations across all runs. For example, a single run for the 8 MiB-16 MiB write-read combination results in 99 writes and 115 reads, each sized at 8 MiB and 16 MiB, respectively. During post-processing of the output runtime data, we keep a running sum of the write and read durations across 100 runs. To find the average write and read durations, and subsuqently the PCIe host write and read bandwidth, for 8 MiB and 16 MiB, respectively, we divide the first running sum by 99,000 and the second by 115,000.

4.8.3 Evaluation Method - CXL Model

The second part of our evaluation is developing an analytical model for CXL that can be used to understand the expected impact of using CXL. This model is built from high-level expectations about the performance of CXL and from understanding the general operation of CXL. It is useful as a way to start exploring and understanding the benefits of using CXL over existing PCIe methods. This evaluation also includes a discussion about the expected changes to the programming paradigms used when writing accelerator code. See 4.8.4 for the CXL Model. See 10.1 for a discussion on how CXL could change the way accelerator programs are written. See 10.2 for CXL model results.

4.8.4 CXL Model

Our CXL model is based on high-level expectations on how CXL will operate from the CXL specification and from various presentations on CXL [5]. In addition to the CXL component of the model, we also include a PCIe model to compare against. CXL utilizes the PCIe 5.0 physical layer infrastructure and the existing method of accelerator communication uses PCIe, so it is important to include a PCIe model along with the CXL model to understand the potential benefits of CXL over the existing PCIe transfer method. For the PCIe portion of the model, we leverage the work of Neugebauer et al. in "Understanding PCIe performance for end host networking" [18]. Currently, we are using PCIe 3.0 with x16 lanes for the PCIe model, since PCIe 3.0 is the latest PCIe version supported in the model and also the version used for both the FPGA and GPU in our system. In our model, we calculate the expected transfer time of n bytes with Equation (9):

$$t_{pcie} = \frac{n}{b_n} + t_s \tag{9}$$

where t_{pcie} is the expected transfer time, n is the number of bytes being transferred, b_n is the expected bandwidth of PCIe when n bytes of data is transferred, and t_s is the expected startup latency for the PCIe transfer. The value of t_s is extracted from Figure 2 in [18] by performing linear regression on the plot and extracting the y-intercept. The PCIe model is able to output the expected PCIe bandwidth for reading, writing, or both. For the creation of this model, we use the bandwidth of both since the assumption is made that the application is pipelined with overlapping reads and writes.

Although there is no publicly available information on the performance of CXL, we can still create simplified models of CXL based on the design of CXL and some performance estimates from CXL slides. As more information about CXL performance becomes available or as hardware is available for benchmarking, the parameters of the CXL model can be updated to provide more accurate insight.

We provide two methods of modeling the performance of CXL. The first method, CXL model 1, is a linear model of expected transfer time of n bytes based on the assumption that the bytes would be moved by $n/cache\_line\_size$ cache misses and each cache miss takes $cache\_access\_time$. The transfer time for this model is therefore given by (10).

$$t_{cxl\_1} = \frac{n}{cache\_line\_size} * cache\_access\_time \tag{10}$$

The cache line size for CXL is 64 bytes and the cache access time is estimated to be 40 ns based on CXL 1.1 Training videos: CXL Overview [6].

The second method, CXL model 2, is based on the assumption that CXL will have less latency than PCIe and will be able to achieve a bandwidth similar to PCIe but with some overhead. Also from the CXL 1.1 Training videos: CXL Overview, the expected bandwidth efficiency of CXL.cache and CXL.mem is 60–90% of PCIe while CXL.io is expected to be similar to PCIe. The CXL model 2 is shown in (11).

$$t_{cxl\_2} = \frac{n}{b_n \cdot cxl\_penalty} + t_s \tag{11}$$

Where t_{cxl_2} is the expected transfer time, n is the number of bytes being transferred, b_n is the expected bandwidth of PCIe when n bytes of data is transferred, $cxl\_penalty$ is the percent of PCIe bandwidth CXL can achieve, and t_s is the expected startup latency of a CXL transfer. The value of t_s is the same as the cache access time used in CXL model 1 and the CXL penalty was chosen to be a pessimistic 60% of PCIe performance.

With the model now able to predict the time required for a data transfer with a particular data size, a simple model of an application's execution time can be created. This simple application model estimates the time required for the execution of a single compute kernel offloaded to an accelerator using PCIe or CXL. This model assumes that the application is fully pipelined with a balanced pipeline, which means that reading, writing, and execution of the kernel occur in parallel and each component takes roughly the same amount of time, as shown in Figure 15. This means that the total execution time for the application can then be computed as shown in Equation (12):

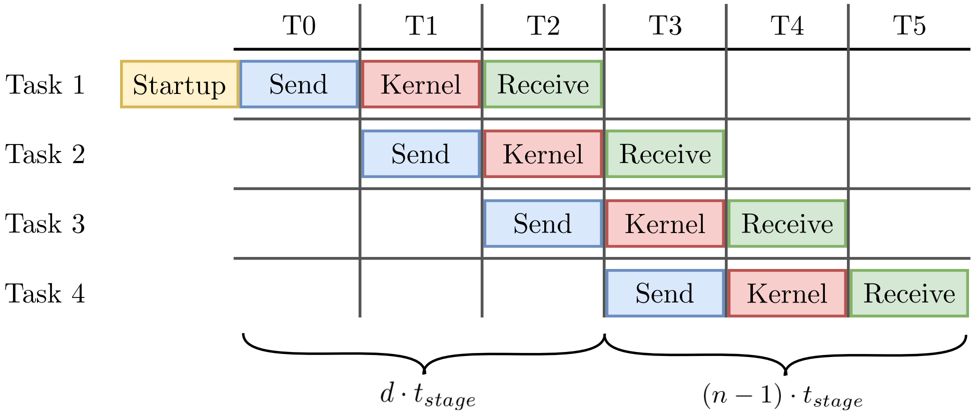


Figure 15: Diagram of the balanced pipeline application execution modeled by the simple application model. In this diagram $d = 3$, $n = 4$, and $s =$ Startup.

$$t_{app} = d \cdot t_{stage} + (n-1) \cdot t_{stage} + s \tag{12}$$

where d is the depth of the pipeline, t_{stage} is the time it takes to execute one stage of the pipeline, n is the number of times the tasks that the pipeline must execute, and s is any additional startup cost for launching the application. t_{stage} is calculated using the respective PCIe or CXL model and s is a configurable parameter. The intuition of the model is that executing a single task will take $d \cdot t_{stage}$ time, since the entire pipeline must be executed before that task completes. However, assuming that subsequent tasks are launched one after the other without delay, one task will be completed after every t_{stage} amount of time. For our data collection, we set s to zero since we assume that the startup cost is much smaller than the application runtime.

5. BENEFITS OF TIGHTER CPU-GPU INTEGRATION

Ever since DOE built its first GPU testbed, users and scientists have opined on the PCIe bottleneck with little application-level quantitative data showing how much it constrains performance. Until recently few GPU vendors had access to high-performance CPU cores while Intel did not have access to high-performance GPUs. As such, CPU-GPU integration within a package was relegated to the realm of idle speculation. However, today, Intel, AMD, and NVIDIA all have access to both high-performance GPUs and high-performance CPUs that could be integrated in a single package. Concurrently, advances in interconnect technologies offer the prospect of much higher CPU-GPU bandwidth.

Without embracing a particular solution, in this section, we investigate the application-level benefits of technologies that provide 10× higher CPU-GPU bandwidths and 10× lower CPU-GPU latencies compared to those observed on PCIe-attached V100s. The former will benefit large host-to-device copies while the latter will benefit small host-to-device copies and CUDA kernel launch overheads. We acknowledge that a 10× shift is large, but note that application-level benefits will be even smaller with technologies that provide less than a 10× (peak) benefit.

5.1 AMREX

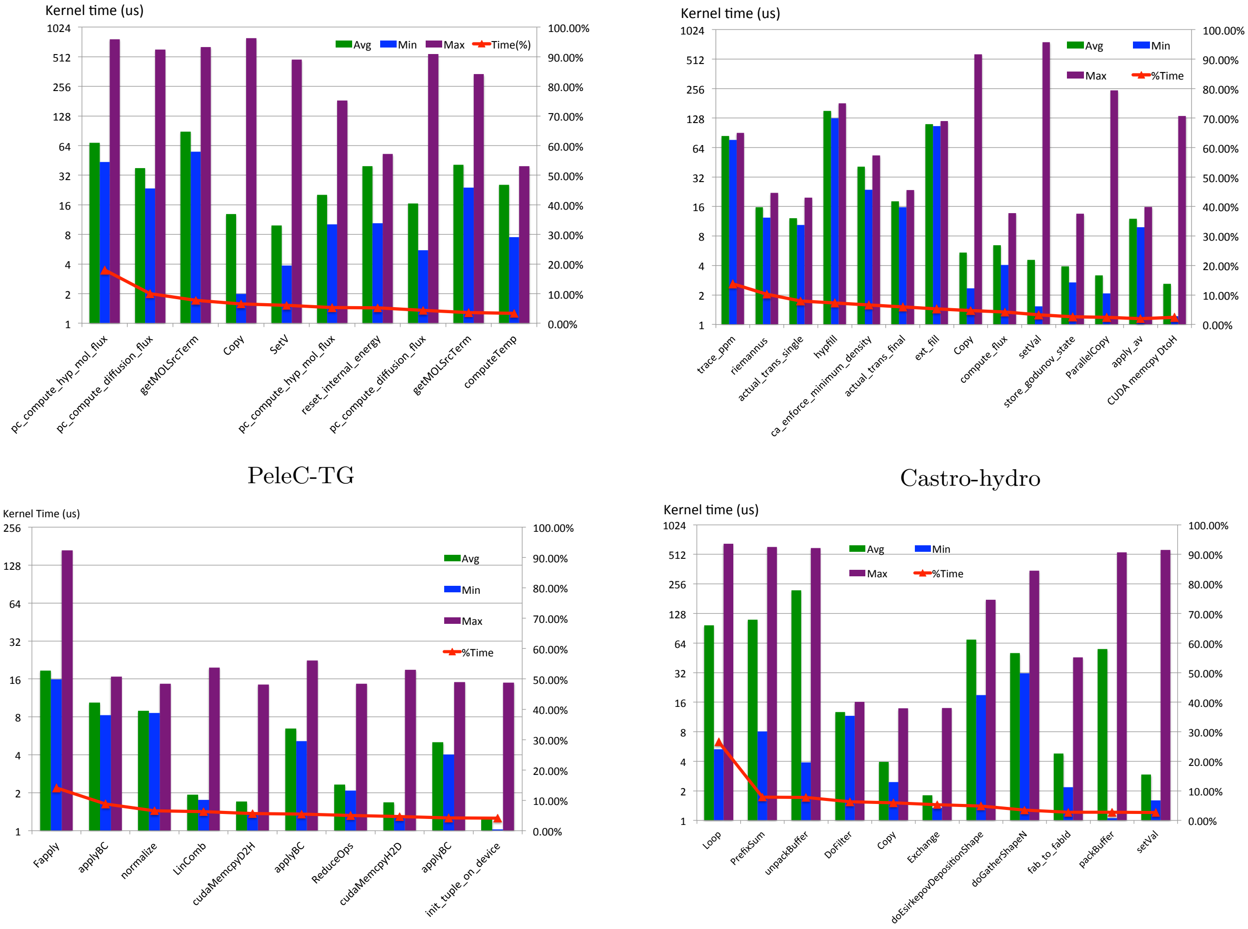
AMReX applications rely on UVM to dynamically migrate data between CPU and GPU (if needed) upon each kernel launch. Some of these applications also move data explicitly using *cudaMemcpy*. As a result, one can expect notable speedup from a more tightly coupled CPU-GPU design due to a lower latency, higher bandwidth connection between CPU and GPU. The speedup can be even more significant on modest data partition sizes (e.g. we use boxes of 32^3 in this study which is commonly seen in practice). To quantify the detailed benefit of the tightly coupled processor design to different AMReX applications, we analyze different costs of the discrete GPU model relative to the total execution time on GPU.

We first evaluate the cost of explicit data movement. Figure 16 presents the execution time of key kernels of 4 AMReX applications. We sort kernels based on their execution time and report kernels until the accumulated time contribution reach 80% of the total execution time. It can be seen that the time spent on cudaMemcpyHtoD and cudaMemcpyDtoH is negligible. For PeleC and WarpX, we don't even see these activities in the top expensive kernels. This leaves us with only two remaining questions for our performance study: cost of UVM's implicit data movement and kernel launch overhead.

Application	Kernel Calls	Launch overhead	UVM
PeleC	1.8e6	0.03%	53%
Castro	1.5e4	0.2%	0.07%
MFiX	7.1e6	18%	0.1%
WarpX	9.8e6	29.4%	0.5%

Table 2: WarpX and MFiX issue many small kernel invocations, explaining for the high kernel launch overhead. PeleC's kernel run for longer time (low launch overhead), but half of kernel time is spent for moving data to and from the host under UVM. Castro uses GPU less intensively (fewer kernel calls), leading to small launching overhead and data transfer across the PCIe with repsect to the total execution time.

Table 2 summarizes the kernel launch overhead and the data movement cost via UVM. As the results show, MFiX and WarpX invokes many small kernels and thus the performance is sensitive to launch overhead. Since these two applications spend 18% and 29.4% for launching kernels, a 10x latency improvement in PCIe connection will result in 1.2× and 1.3× speedups. With PeleC, we observe significant explicit data movement (53% total execution time). A 10× bandwidth improvement should speedup this application by 1.9×. Castro does not use GPU extensively, leading to small launch overhead and data movement when comparing to the total execution time (including CPU time). Thus, it is unlikely that improvements on PCIe latency and bandwidth can lead to higher performance.



PeleC-TG

Castro-hydro

Mfix-extern pipe

WarpX-plasma acceleration

Figure 16: Performance of important kernels and explicit data movement operations (cudaMemcpyH2D and cudaMemcpyD2H) invoked by AMReX applications. These activities are sorted based on their contribution to the total execution time. We observe modest explicit data movement.

5.2 BERKELEYGW

The nvprof profile of BerkeleyGW's Epsilon module is shown in Table 3. The largest potential speedup for integration is from the GPU memory management functions (within $t_{CGcontrol}$); a modest number of longer-running (O(ms)) calls make up about 18% of the walltime. Kernel launch times are the second leading fraction of $t_{CGcontrol}$, and account for 17% of the total Epsilon runtime. In this benchmark, a the vast majority of GPU kernel calls have runtimes of O(10 μs) and average launch times of 11 μs. While the impact of kernel launch times could conceivably be mitigated at the application level by consolidating many calls to the same function reduced kernel launch latency is a major potential benefit of tighter CPU-GPU integration. Tighter integration through improved on-node interconnect (e.g. PCIe) performance could decrease t_{CGdata}, and increase application performance by 10%. Supposing that CPU-GPU integration would improve $t_{CGcontrol}$ and t_{CGdata} by a factor of 10 (i.e. the λ parameters of Equation 4 are set to 10), integration would improve overall performance by 1.9×.

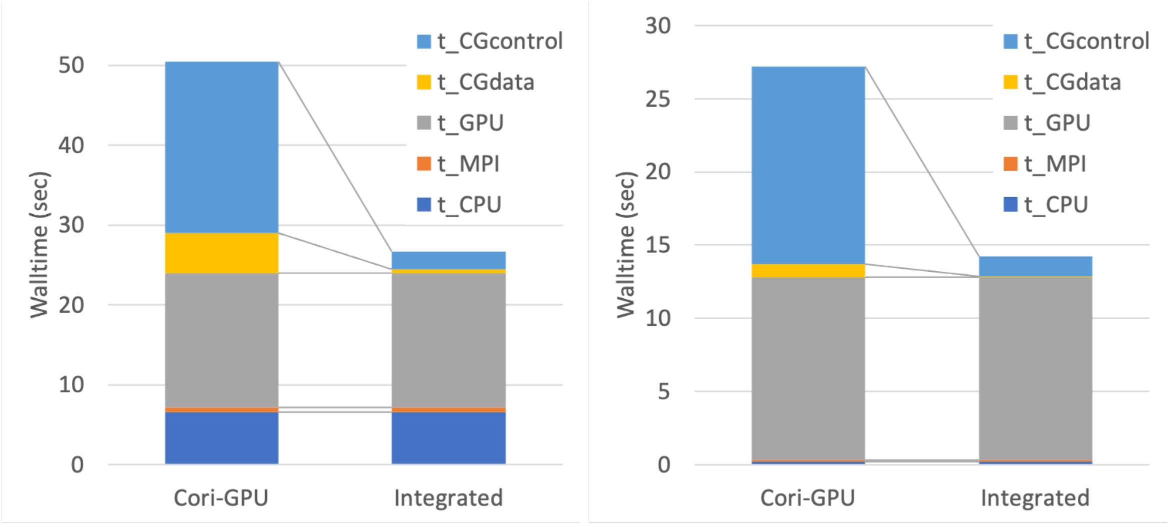
The nvprof profile of the Sigma module is shown in Table 4. For Sigma, t_{CGdata} contributes little to total runtime. Within $t_{CGcontrol}$, a modest number GPU memory management takes 26% of the total runtime, and stream synchronization takes nearly 20%. Accelerating these functions by a factor of 10 would improve overall Sigma performance by 1.9×, as illustrated in Figure 17b.

Category	Function Name	Time	Calls	Avg	%Total
t_{GPU}	composite_2way_fft	7.37	482622	15 us	14.6
	volta_zgemm_64x32_nt	3.76	32	117 ms	7.5
	pairwise_mult_vect	3.76	160072	24 us	7.5
	get_from_fftbox	1.44	160072	9.0 us	2.9
		16.79	963457		33.3
$t_{CGcontrol}$	cudaLaunchKernel	8.47	802838	11 us	16.8
	cudaFree	7.72	1272	6.1 ms	15.3
	cudaMalloc	1.85	1262	1.5 ms	3.7
	cudaHostAlloc	1.06	749	1.5 ms	2.1
	cudaStreamSynchronize	1.03	161701	6.3 us	2.0
	misc. cuda APIs	1.97	1305226	1.5 us	3.9
		22.10	2273048		43.8
t_{CGdata}	[CUDA memcpy HtoD]	3.65	887	4.1 ms	7.2
	[CUDA memcpy DtoH]	1.40	160104	8.8 us	2.8
		5.1	160991		10.0
t_{CPU}		6.54			12.9
t_{MPI}		0.66			1.3
t_{total}		50.5			100.0

Table 3: NVprof profile of BerkeleyGW::Epsilon running on two Cori-GPU nodes with one MPI rank per GPU (16 ranks total), and five OpenMP threads per rank.

Category	Function Name	Time	Calls	Avg	%Total
t_{GPU}	cuda_sigma_gpp_kernel_n1_in	11.34	32	354 ms	41.7
	composite_2way_fft	0.83	38412	22 us	3.1
	pairwise_mult_vect	0.31	6400	48 us	1.1
		12.48	44844		45.9
$t_{CGcontrol}$	cudaFree	7.24	276	26 ms	26.6
	cudaMalloc	0.36	228	1.6 ms	1.3
	cudaStreamSynchronize	5.29	6490	816 us	19.5
	cudaLaunchKernel	0.34	57680	5.9 us	1.2
	cudaGetDeviceProperties	0.29	40	7.3 ms	1.1
		13.52	64714		49.7
t_{CGdata}	[CUDA memcpy HtoD]	0.54	6561	82 us	2.0
	[CUDA memcpy DtoH]	0.36	6528	54 us	1.3
		0.90	13089		3.3
t_{CPU}		0.19			0.7
t_{MPI}		0.12			0.4
t_{total}		27.21			100.0

Table 4: NVprof profile of BerkeleyGW::Sigma running on one Cori-GPU nodes with one MPI rank per GPU (8 ranks total), and five OpenMP threads per rank.



(a) Epsilon

(b) Sigma

Figure 17: Measured and modeled walltime breakdowns for BerkeleyGW. Profile measurements were obtained on Cori-GPU. Model predictions for a hypothetical integrated system scaled the $t_{CGcontrol}$ and t_{CGdata} contributions by 1/10.

5.3 GTCP

GTCP uses an explicit cuda-based offload to leverage accelerator architectures. Time consuming routines such as particle push, charge deposition are executed on the GPU. The particle shift between ranks involves execution on the CPU and the GPU. As such, the routines are typically coarse grained and the latency component of $t_{CGcontrol}$ component is typically small.

Table 5 summarizes the model parameters, presented in Section 4.2 for GTCP. To estimate the impact of tighter CPU-GPU implementation, based on the model presented in Section 4.2, we reduced the latency of control flow by up to 10×. The impact on $t_{CGcontrol}$ is minimal (<%1). The impact of increasing the bandwidth by up to 10× is roughly 20%.

5.4 SPTRSV

The modeling results of SpTRSV is shown in Figure 18b. The input matrix S1 comes from M3D-C1, a fusion simulation code used for magnetohydrodynamics modeling of plasma [11]. Li matrix is from SuiteSparse Market Collection. The matrix is first factorized via `SuperLU_DIST` with METIS ordering for fill-in reduction [12]. The resultant lower triangular matrix is used with the proposed critical path model. The factored S1 matrix has 9,827 supernodes, 880 million nonzeros in double-precision, and 388 DAG levels. The factored Li matrix has 332 supernodes, 518 million nonzeros in double-precision, and 188 DAG levels. The overall sweet spot for both S1 and Li is six GPUs within a node. From the model, we see that the numbers of messages for S1 on the critical path are very similar: 667 messages (six GPUs) and 706 message (twelve GPUs), but the network bandwidth decreases as more nodes are used. Ultimately, the resultant low inter-node NVSHMEM throughput makes the run time of S1 increase when using more than one node.

Figure 18a shows that the intra-socket NVSHMEM (`nvshmem_double_put_nbi_block`) bandwidth outperforms intra-node and inter-node by 1.2× and 3.9× on Summit (6 GPUs per node). The NVSHMEM intra-socket and intra-node implementation is via GPU p2p memory access. However, for the inter-node communication, NVSHMEM only uses one single thread in a thread block to issue an RMA write operation to the destination GPU over InfiniBand. Therefore, we see a big slowdown in NVSHMEM inter-node communication.

According the Study of Host and Device-initiated Approaches [10], if we compare a CPU-initiated NVSHMEM put (2us) to a GPU-initiated NVSHMEM put (6.5us), GPU-initiated NVSHMEM put is 3.25× slower. The extra overhead comes from the synchronization in GPU memory. In NVSHMEM, the GPU-initiated communication operations over Infiniband offload work to progress threads running on the CPU. When a GPU thread issues a

Category	Function Name	Time (s)	Calls	Avg (ms)	%Total
t_{GPU}	permute_particles_zelectron_ph1	8.64	960	9.00	7.09
	update_particle_zelectron_ph1	8.55	960	8.90	7.04
	gpu_push_kernel	6.32	480	13.17	5.19
	update_particle_zelectron_ph2	2.33	960	2.42	1.90
	permute_particles_zelectron_shift_ph2	2.32	936	2.48	1.907
	DeviceRadixSortDownsweepKernel	1.40	2880	0.49	1.14
	DeviceRadixSortDownsweepKernel	0.89	1920	0.46	0.73
	calculate_sort_keypair_shift	0.81	456	1.78	0.68
	kernel_agent_thrust	0.45	1920	0.24	0.38
	DeviceRadixSortUpsweepKernel	0.35	2880	0.12	0.30
	Others	0.99	7744	0.13	0.82
		33.05	22096		27.17
t_{malloc}	cudaMalloc	7.33	975	7.51	6.00
	cudaFree	4.03	975	4.13	3.28
	cudaHostAlloc	3.20	7	456.50	2.60
		14.55	1957		11.1
$t_{CGcontrol}$	cudaEventSynchronize	28.43	5784	4.91	23.39
	cudaMemcpyAsync	10.94	6720	1.63	9.00
	cudaStreamSynchronize	9.07	960	9.45	7.47
	cudaDeviceSynchronize	3.29	1920	1.71	2.72
	cudaLaunchKernel	0.24	22104	0.01	0.17
	misc. cuda APIs	16.91	24643		13.76
		68.88	62131		56.51
t_{CGdata}	[CUDA memcpy DtoH]	10.71	1464	7	8.8
	[CUDA memcpy HtoD]	10.70	8188	1	8.7
		21.41	9652		17.6
t_{CPU}		42.52			34.96
t_{MPI}		24.65			20.27
t_{total}		121.63			100.0

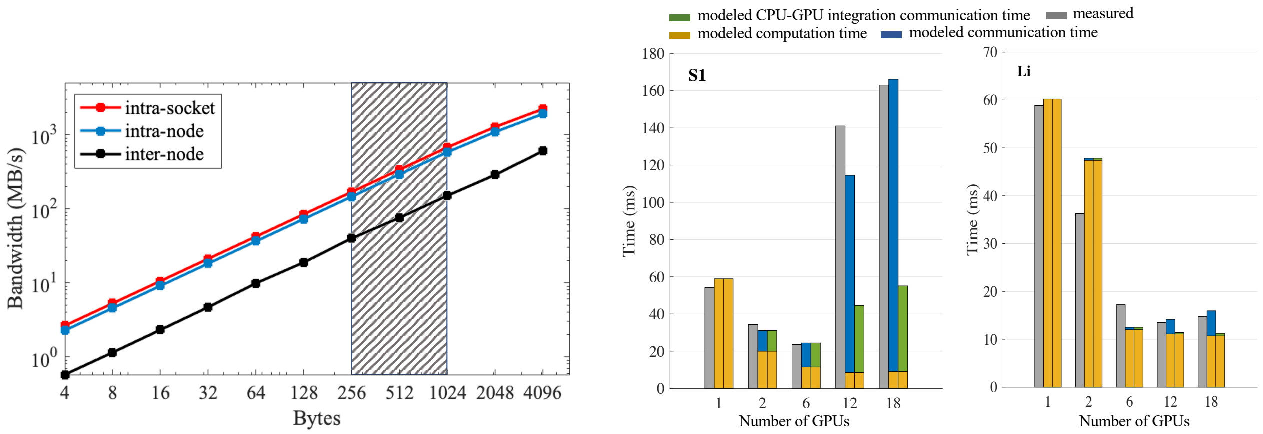
Table 5: Profile of GTCP running on two Cori-GPU nodes with one MPI rank per GPU (16 ranks total), and eight OpenMP threads per rank.

communication operation, it first issues a global memory fence to flush all modified states from the L2 cache to GPU global HBM2 memory. Only after it finishes this global memory fence can it perform an enqueue operation to the queue associated with its CPU proxy thread. Once this write is completed, the CPU progress thread can dequeue the metadata written by the GPU thread and prepare and send the corresponding Infiniband request to the NIC. As such, if CPU-GPU integration can contribute 10× better CPU-GPU latency and bandwidth, it can reduce the synchronization overhead in a GPU-initiated NVSHMEM put, thus making NVSHMEM performance 2.6× faster (2us CPU-initiated + 0.5us synchronization overhead). Therefore, to estimate the CPU-GPU integration performance of SpTRSV, we adjust the inter-node NVSHMEM performance by 2.6× faster.

Figure 18b presents the modeled results of S1 and Li matrix. S1 matrix still gets its best performance at six GPUs. However, when it comes to inter-node, CPU-GPU integration can improve the performance by 3×. Matrix Li achieves its best performance at eighteen GPUs with a speedup of 5× over a single GPU with the help of CPU-GPU integration. As such, for the matrices like S1 which has a strong data dependency (communication intensive), although CPU-GPU integration can improve inter-node performance, performance still stops scaling at six GPUs. That means in order to have a better scaling performance, NVSHMEM still requires an further optimized performance via infiniband. For the matrices like Li (less data dependencies, computation intensive), CPU-GPU integration can bring potential performance benefits by 1.6× using eighteen GPUs over the best baseline using six GPUs.

5.5 HACC, LAMMPS, OPENMC, AND RODINIA NW

Figure 10 and 11 tells us the practical range of offloading bandwidth and latency on the current generation of HPC-class GPUs. Calibrating the Agochart model with these numbers, we projected the speed up based on application trace results, targeting a GPU with 1,024 execution units runs at 1.8 GHz, varying the simulation knobs of AgoChart with the latency between 1 and 1,000 µsec and the bandwidth between 1 GB/s and 64 GB/s. While 1,000 usec is unrealistic for a single GPU/single-process case, such higher latency may occur on multi-GPU



(a) NVSHMEM (thread block) bandwidth using two GPUs on Summit. The shadowed stripe highlights the typical message size in SpTRSV of 256 bytes to 1,024 bytes.

(b) The total SpTRSV time equals the accumulation of the stacked bars

Figure 18: The overall SpTRSV performance is limited by the resultant low inter-node NVSHMEM throughput. Tighter CPU-GPU integration would bring potential benefit of higher GPU-GPU network performance, and the SpTRSV would be improved by a factor of 5× using eighteen GPUs.

systems or single GPU shared by multiple processes. Similarly, 1 GB/s is certainly far below the lower end of available bandwidth for an isolated environment where only a single task is running, however, when multiple applications compete a set of shared resources, this could occur.

Figure 19, 20, 21 and 22 show the performance projection results on HACC, LAMMPS, and OpenMC in a 3D plot. The results show that HACC incurs a 20-30% performance penalty around the realistic PCIe BW (8-16GB/s). LAMMPS is sensitive to higher latency. OpenMC is primarily sensitive to latency. It showed negligible performance penalty around a realistic PCIe BW (8-16GB/s), which also takes into account software stack overhead. RODINIA [4] NW is not an ANL application, but we present this as an example of an unoptimized workload, which issues too many offloading requests and its offloading kernel completes in an extremely short duration of time (e.g., 30-40 usec).

Our definition of CPU-GPU tighter integration is to lower the initiation latency of data transfer and kernel offloading on a GPU between host and GPU. We chose 100 µsec initiation latency as our baseline reference and one with 10 µsec initiation latency for our target while we chose 16 GB/s effective data transfer bandwidth between host and GPUs for both baseline and target, which approximately represents the upper-bound practical performance that an application that runs on an A100 GPU can observe. The numbers in the CPU-GPU integration column in Table 13 are calculated by the performance gain of the target and that of the baseline reference.

5.6 ESTIMATION OF THE CPU-GPU INTEGRATION BENEFITS WITH SSI FOR SW4LITE, MINIVITE, QUICKSILVER, AND LAGHOS

To predict the effect of tighter integration or decoupling of CPUs and GPUs on different applications, we utilize our SSI simulator and vary the CPU-GPU connector link properties as well as modify the number of data transfer channels across the hosts and GPUs. We change the current latencies of the link, as depicted in Figure 23.

Since the simulation infrastructure uses propagation and transmission delay for their internal calculations, we used the propagation delay (which is a composite parameter that includes link latency and the length of the link) while fixing the number of lanes and transmission delay. We approximate the possible idealized link lengths by using the speed of light (in vacuum) and evaluate links from 0 (theoretical bound) to 16 cm. We have found that the maximum length of PCIe4 is around 20-30 cm. We denote the resulting simulation time based on our minimum link length by α and the maximum time based on the longest link by ω. We calculated the simulated time using the current link properties (validated in Figure 5) and denote it by δ. Based on our ideal length calculations, we find the length of Nvlink2 and PCIe4 to be approximately 1 cm and 2 cm respectively. With

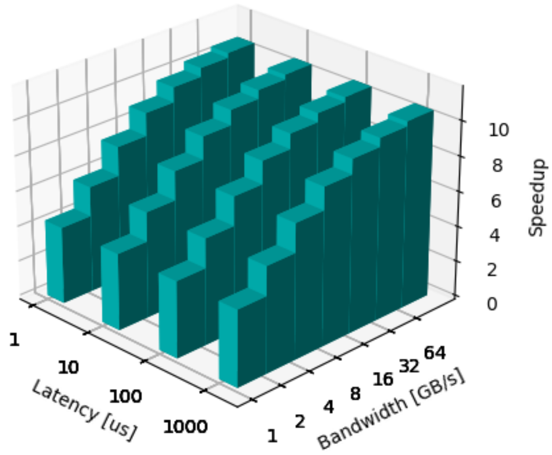


Figure 19: AgoChart Projection: HACC

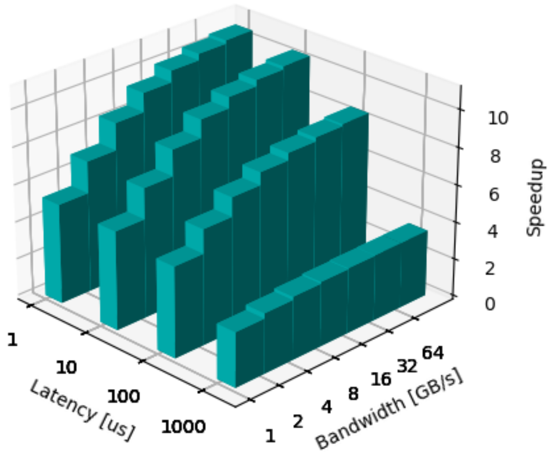


Figure 20: AgoChart Projection: LAMMPS

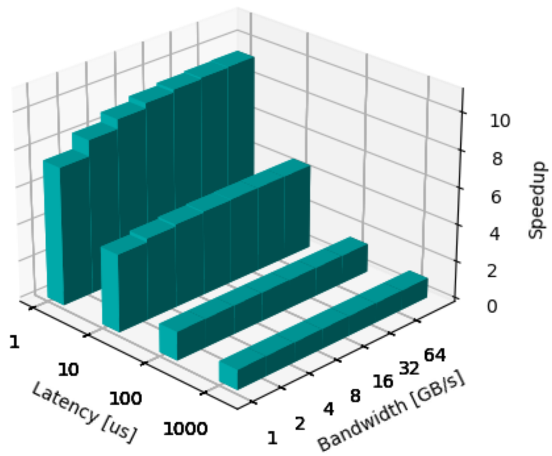


Figure 21: AgoChart Projection: OpenMC

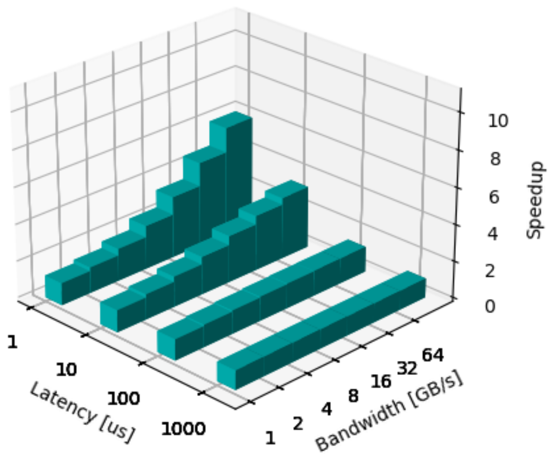


Figure 22: AgoChart Projection: Rodinia NW

these simulation times, we define the $IntegrationBenefit = \delta/\alpha$ and $IntegrationPenalty = \omega/\delta$. For each of the applications we simulate, we take the largest kernel from a given workflow and simulate its execution time and data transmission.

5.6.1 SW4Lite

We set the parameters of SSI to simulate the kernel rhs4_v2 with the total number of bytes transmitted over PCIe and NVLINK (60093 bytes). We simulated 100 kernels of the same kernel, the same number of kernels invoked and report the simulation results in Figure 24. For the best time, using NVLink2, we see that the benefits of integration are very low (around 1.009x). When using the PCIe4 links, we obtain a better benefit overall (since the link itself is slower) of 1.027x. We find that there is little benefit for aggregation, since the data transferred per message is low (i.e. approximately 60KB).

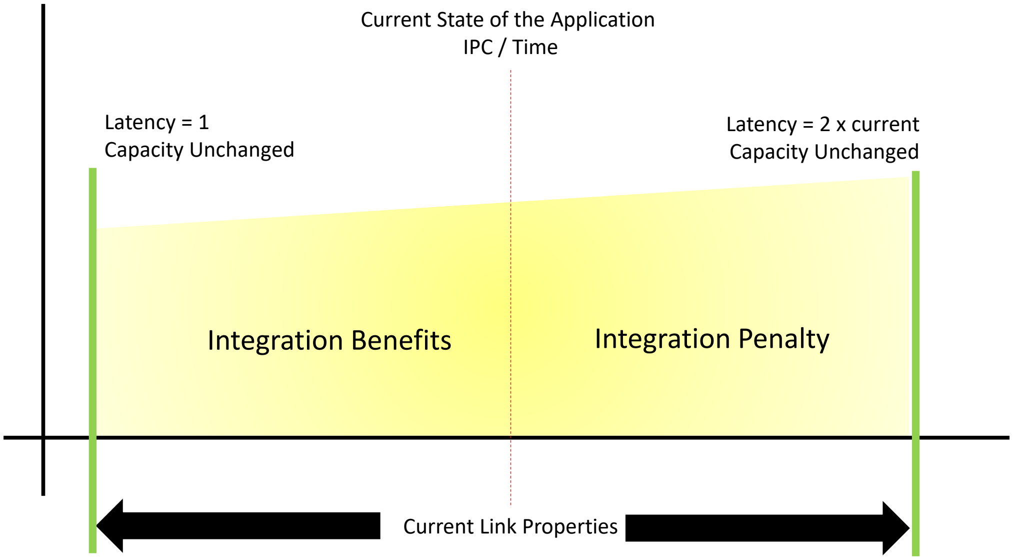


Figure 23: Procedure to simulate benefits and detriments of the integration effort

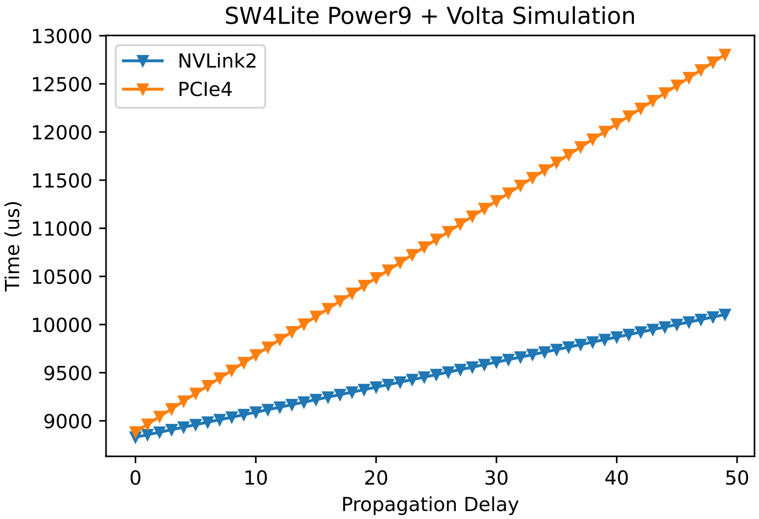


Figure 24: SW4Lite Simulated time for rhs4_v2, when varying propagation delay of the link between CPU and GPU.

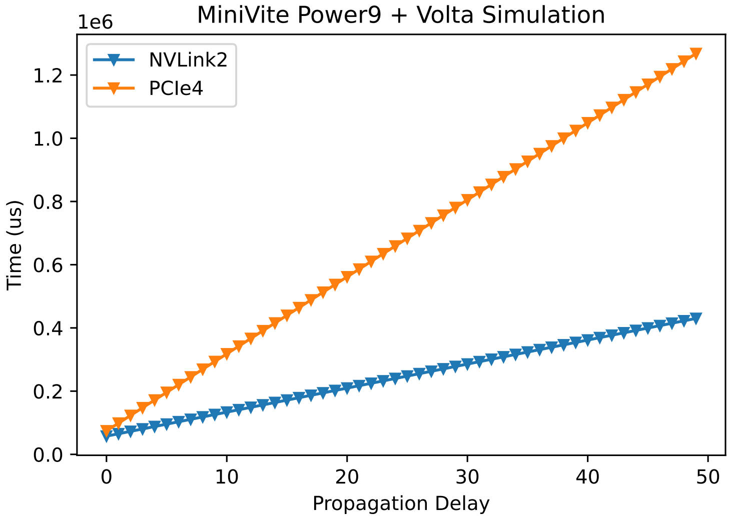


Figure 25: MiniVite simulation time when varying propagation delay of the link between CPU and GPU.

5.6.2 MiniVite

We set the parameters of SSI to simulate the most computational expensive kernels of MiniVite with the total number of bytes transmitted over PCIe and NVLINK (9693535 bytes). We simulated 195 kernels, the same number of times that the kernel was invoked. We report the simulation results in Figure 25. For the best time, using NVLink2, we see that the benefits of integration are still low (around 1.40x). When using the PCIe4 links, we obtain better benefit overall (since the link itself is slower) of almost 2x (1.98x). For this miniapp, while low, we observe better advantage compared to Sw4Lite since the size of the transferred messages are larger. This impact is of course observed because of the slower technology (propagation delay) of PCIe4.

5.6.3 QuickSilver

We set the parameters of SSI to simulate the cycleTracking kernel with the total number of bytes transmitted over PCIe and NVLINK (45966 bytes). We simulated 45 kernels, the same number of times that the kernel was invoked and we report the simulation results in Figure 26. For the best time, using NVLink2, we see that the benefits of integration are very low (around 1.01x) almost as bad as the ones for SW4Lite. When using the PCIe4 links, we obtain a better benefit overall (since the link itself is slower) of 1.027x which is the same as SW4Lite. This is not surprising since both of the miniapps share similar communication pattern. Similar to Sw4Lite, we observe relatively insignificant impact of tighter integration, since the amount of data sent over the links between the CPU and the GPU is small.

5.6.4 Laghos

We set the parameters of SSI to simulate the kernels of Laghos with the total number of bytes transmitted over PCIe and NVLINK (921530 bytes). We simulated 57253 kernels, the same number of times that the kernel was invoked and we report the simulation results in Figure 27. For the best time, for both NVLink2 and PCIe, we observed that the benefits of tighter integration to be around 4x. For this application, we observe noticeable impact on the simulated execution time as the lane lengths decreased. This is due to the fact that Laghos workload consists of larger number of kernel calls and relatively larger messages.

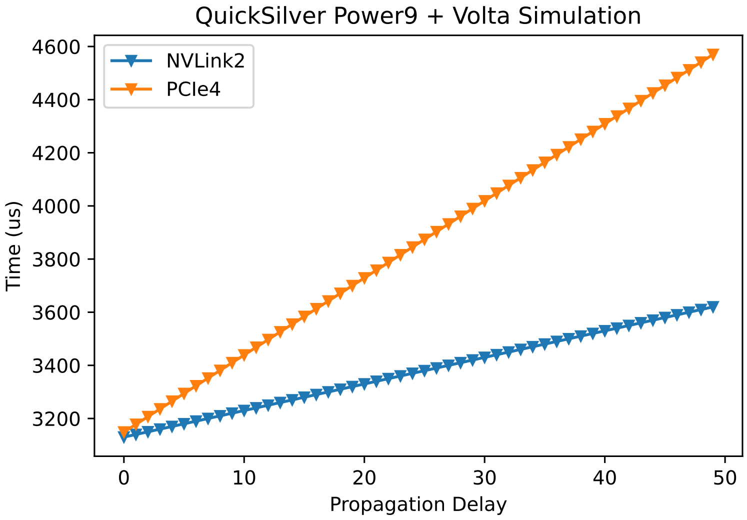


Figure 26: QuickSilver Simulated time for cycleTracking kernel when varying propagation delay of the link between CPU and GPU.

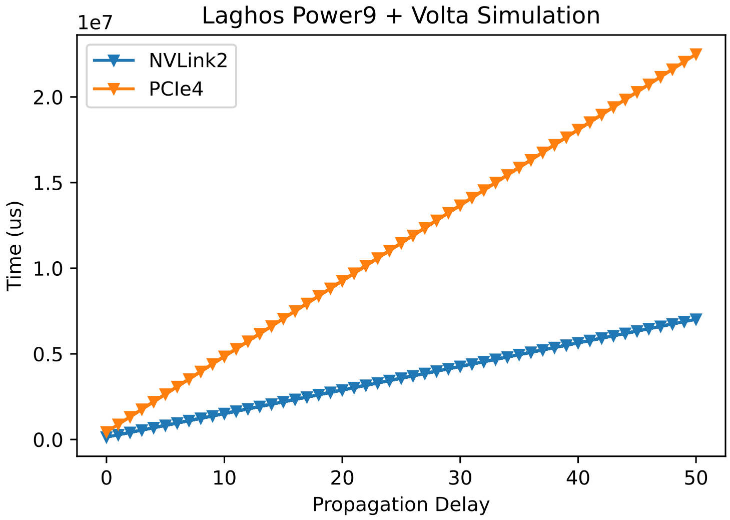


Figure 27: Laghos Simulated time when varying propagation delay of the link between CPU and GPU

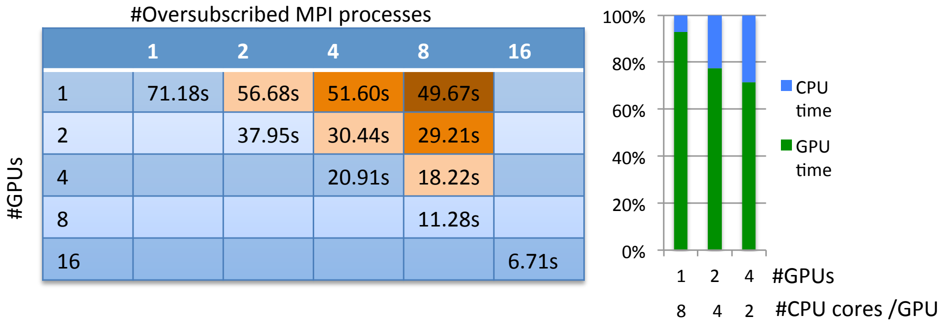
6. CPU:GPU RATIO

Systems based on discrete GPUs are free to vary the number of GPUs per CPU socket as well as (within available SKUs) the number of CPU cores per CPU socket. This has enabled system architects to scale systems to maximize workload performance for a given cost. Unfortunately, when designing systems that integrate CPUs and GPUs into a single package, architects are forced to provide a limited number of SKUs constrained by not only power and pinout, but also the fear of overwhelming customers with too many SKUs. As such, it is imperative DOE provide quantitative data as to the appropriate CPU core:GPU ratio for its workload. In this section, we employ a variety of previously discussed methodologies to determine the number of Skylake-equivalent cores per V100 GPU that attains 90% of the potential application performance. The 90% is arbitrary but enforced to avoid asymptotic gains from exponentially increasing CPU:GPU ratios. CPU:GPU ratios greater than 1:1 can arise from the ability to employ OpenMP, the ability to have multiple MPI processes share each GPU, or artifacts of the communication runtime.

6.1 AMREX

For AMReX applications we map multiple MPI processes to each GPU. The table in Figure 28 lists execution time of CASTRO with multiple CPU core-GPU ratios. As we can see the performance benefit of exploiting host parallelism is notable. Since Cori limits the number of MPI processes per node to 8, we can't try with larger over-subscription factors. However, the time distribution in the bar chart of Figure 28 shows that with 8 MPI processes per GPU, there is almost nothing left on the host.

Figure 29 the results on other applications using the same methodology. It can be seen that these application favor a smaller CPU-GPU ratio than CASTRO. In particular, WarpX performance stops improving at 4 CPU cores while PeleC stops earlier at 2 cores and MFIX does not benefit from using more than 1 core.



#Oversubscribed MPI processes

#GPUs	1	2	4	8	16
1	71.18s	56.68s	51.60s	49.67s	
2		37.95s	30.44s	29.21s	
4			20.91s	18.22s	
8				11.28s	
16					6.71s

Figure 28: Castro's execution time on the host is significant, but can be parallelized with multiple MPI processes that share the same GPU. However, it is unlikely that the speedup can be further improved with more than 8 CPU cores per GPU

6.2 BERKELEYGW

The optimal CPU-GPU ratio for BerkeleyGW was determined using the method described in Section 4.3. These tests were performed on Cori-GPU and used the minimum number of nodes needed to meet the job memory requirements, and every node used all eight of the available V100 GPUs; two nodes (16 GPUs) were used for Epsilon, and one node (8 GPUs) were used for Sigma.

The MPI scaling of both Epsilon and Sigma was non-ideal and was inconsistent with the model in Equation 5. This is illustrated by Figure 30, which shows that the compute time (i.e. Total - MPI) *increases* with the number of MPI processes per GPU. This is most likely due to overheads from the MPS service. (Future generations of NVIDIA GPUs will have "Multi-instance GPU" features that might alleviate this MPI scaling issue.) This problem forced a simplification of the procedure. Namely, we set c_{MPI} and c_{hybrid} to zero, and fit c_{seq} and c_{OMP} to the subset of data that used only one MPI process per GPU.

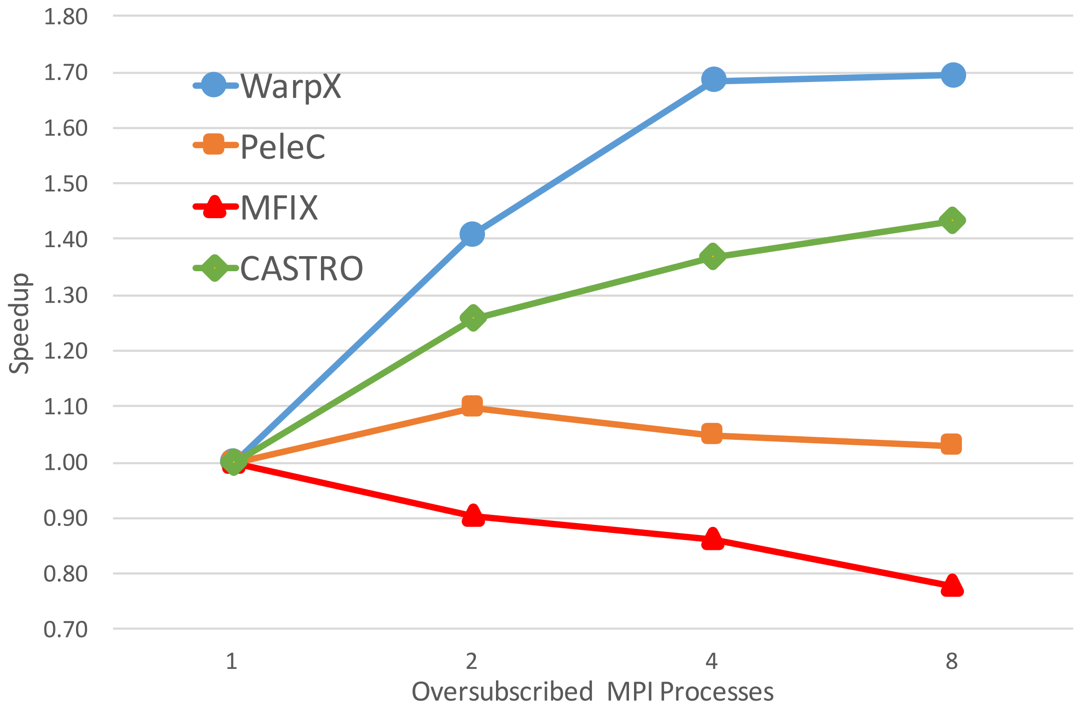


Figure 29: Except for MFIX, we observe performance improvement on other AMREX applications by exploiting host parallelism with multiple CPU cores

The fitting parameters are show in Figure 31a. The t_{MPI} value is the average observed from all runs. There was little run-to-run variation in t_{MPI}, so this choice had limited impact. The sole exception occurred for Sigma: t_{MPI} was 40% larger when using only one OpenMP thread - all other thread counts had the same t_{MPI}. We used the average of the multi-threaded t_{MPI} measurements and accounted for this choice during the optimization of Sigma's CPU-GPU ratio by adding the constraint $n_{OMP} \geq 2$.

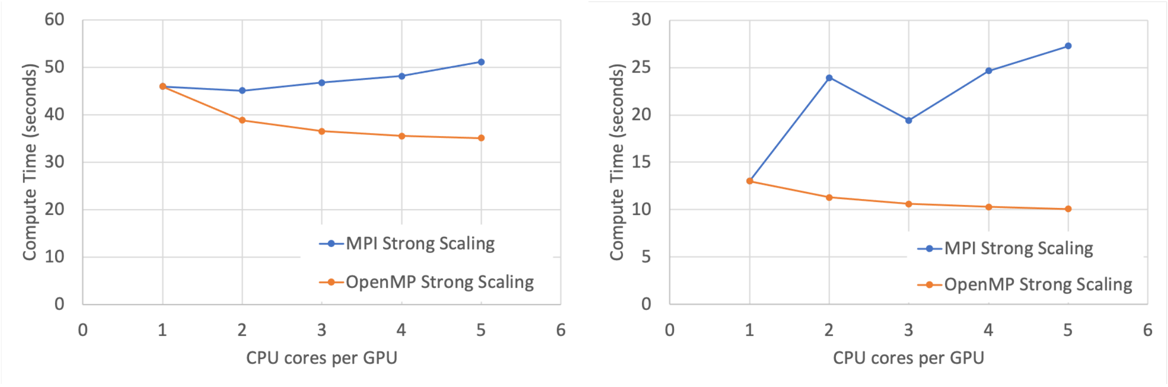
The modeled performance of Epsilon and Sigma are shown in Figure 31b. The equivalent of 3.5 Skylake cores per V100 GPU are needed to obtain 90% of the Epsilon model's asymptotic value, and 2.0 cores are needed for Sigma. Noninteger values are possible because the results stem from an analytical model. In practice, these must be rounded to an integer number of cores. The lower bound for n_{OMP} was the determining factor for Sigma.

6.3 GTCP

To estimate the CPU to GPU ratio that provide 90% efficiency of asymptotic performance, we find the model parameters, described in Section 4.3.2, which fits the measured scaling behavior as we increase the core count per GPU. Table 6 provides the model parameters for key routines. Figure 32 shows the measured scaling behavior vs. modeled one. We use the model to estimate the performance as number of cores reaches infinity. At 90%, the number of cores per GPU varies per routines, as shown in Figure 33. In general, three to six cores are enough to hit the 90% efficiency mark for most kernels. The whole application requires roughly three cores per GPU.

Parameter	Total	Charge	Push
α	121.89	9.38	12.43
β	137.99	45.81	74.47
λ	0.766	0.627	0.84

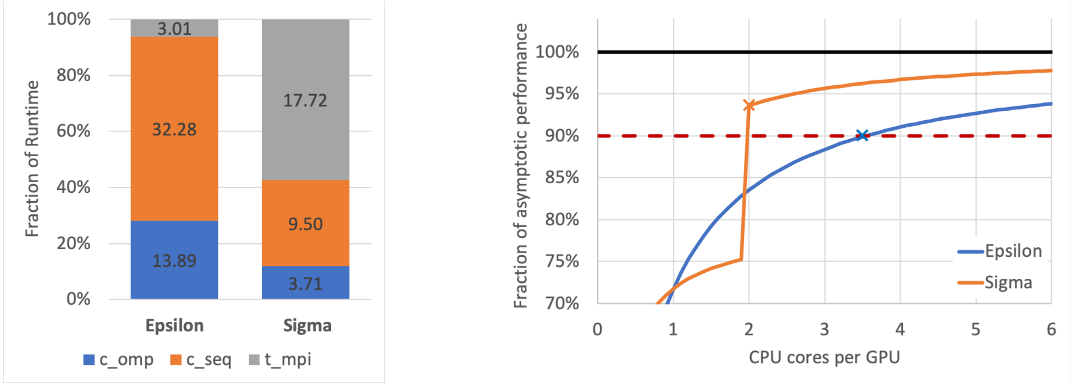
Table 6: Parameters for modeling the GTCP scaling with the number cores per GPU.



(a) Epsilon

(b) Sigma

Figure 30: MPI and OpenMP strong scaling of the compute time (Total -MPI) for BerkeleyGW with a constant number of GPUs.



(a) Model parameters

(b) Model evaluation.

Figure 31: CPU-GPU ratio optimization for BerkeleyGW. (a) Breakdown of the modeled runtime $n_{core} = 1$. The values of the c parameters are printed on the printed on the corresponding bar. (b) Model runtime predictions vs n_{core}. The × marks indicate the smallest values that exceed 90% of the asymptotic performance.

6.4 SPTRSV

The CPU:GPU ratio is 2:1 in SpTRSV because NVSHMEM also launches a CPU proxy thread in addition to the MPI process. So there are two CPU cores per NVSHMEM rank (GPU). That 2:1 ratio can not be changed at the moment because an NVSHMEM rank must have exclusive access to the GPU to to guarantee deadlock-free progress, completion and correctness.

6.5 CONCURRENCY ANALYSIS FOR SW4LITE, QUICKSILVER, MINIVITE, AND LAGHOS

Using our NVDAG tool, we first analyze the amount of concurrency in the existing code base (in terms of the number of concurrently-scheduled kernels in different CUDA streams) for four different applications. This analysis helps us to understand whether the applications require additional GPU resources (and will subsequently benefit from high-capacity inter-GPU links or tighter GPU-GPU integration) or whether the kernels are short-lived, and their occupancy profiles do not mandate additional resources. By representing the concurrently-executed kernels and their dependencies in a mini-app as a DAG, we found out that the applications under study do not take much advantage of the concurrency across GPUs. Even when multiple streams are used (as in the case of SW4Lite and MiniVite), there are either synchronization statements across the phases which serializes the runs

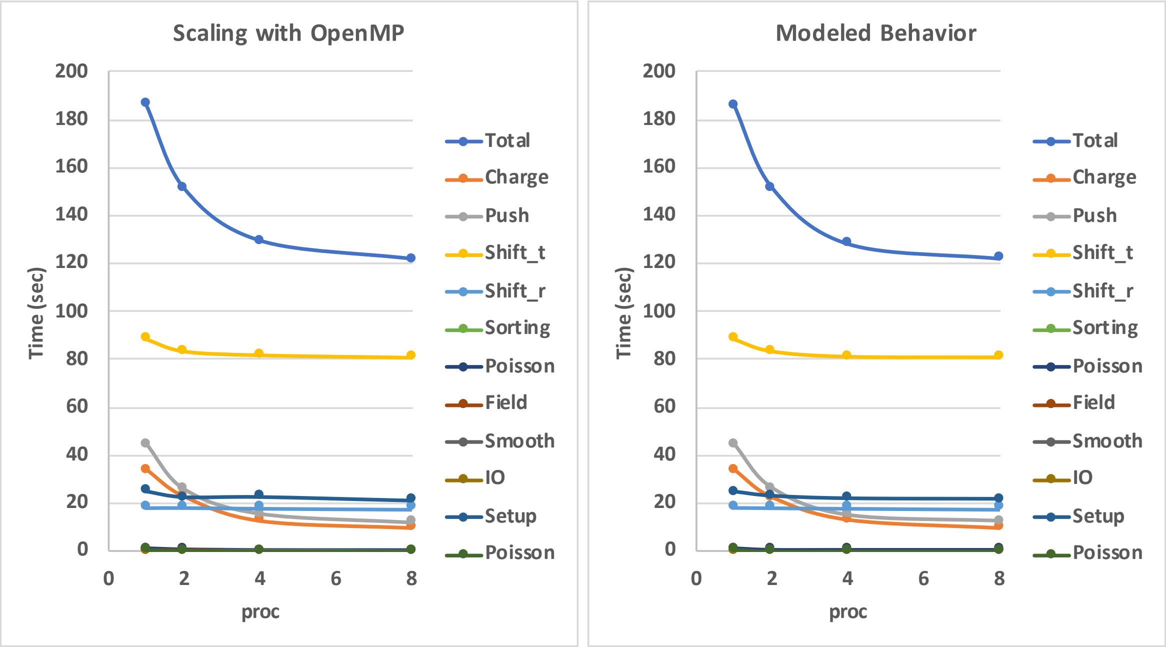


Figure 32: Measured performance as we increase the number of cores per GPU. We used model in section 4.3.2 to estimate the asymptotic scaling behavior as we increase the number of cores per GPU. Both measured and modeled behavior highly correlate.

	Memory(bytes)	Time (us)	Num Calls
MiniVite	9693535	50022	195
QuickSilver	45966	3120	45
SW4Lite	60093	8804	100
Laghos	921530	52	57253

Table 7: Simulation Parameters for our workloads

across the streams (refer to Figure 34a and 34b) or they only use a small number of streams (3 for SW4Lite and 2 for MiniVite). In the case of QuickSilver, every kernel is followed by device synchronize statements so all the inter GPU concurrency collapses to one GPU. Thus, we did not generate graphs for this workflow (since they would collapse to a single line). In the case of Laghos, the workflow only uses one stream so no inter-kernel concurrency is expressed.

Since we have observed that the mini-apps under consideration do not have sufficiently large, concurrently scheduled workload to keep more than one GPU occupied, we only analyze the single GPU runs only for this milestone since multiple GPU runs for these applications are not really taking full advantage of multiple streams/GPUs. Examples of the resulting DAGs (long series of computational nodes with shallow branches of computations) can be seen in Figure 35a and 35b.

We leverage the in-depth profiling information collected in previous milestones to understand the application's characteristics and behavior. This allows us to simulate and predict certain parts of the applications. First, we present the characteristics of interests for each application analyzed together with any ancillary information to explain our findings.

First, we compare the difference between the runtime of the kernels executed on CPU and the runtime of the kernels executed on the GPU to see the advantage of one over the other. Next, we analyze the ratio of the GPU kernel execution time with the CPU execution time. We also record the amount of GPU utilization by each of these kernels.

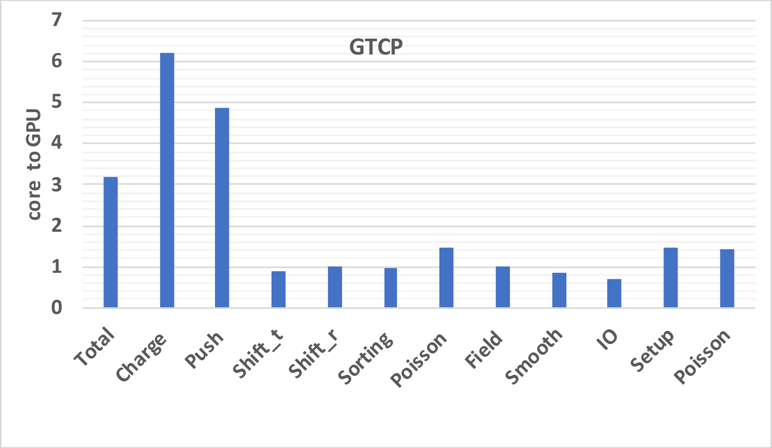
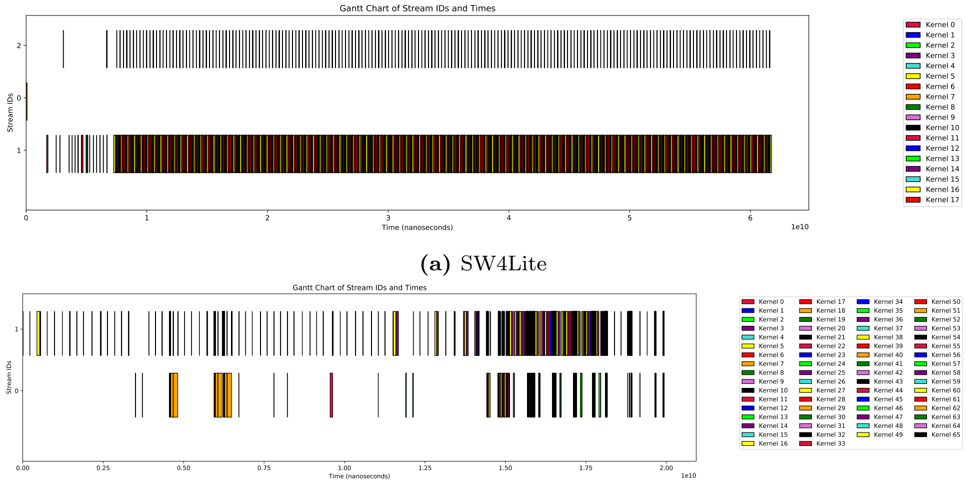


Figure 33: The CPU cores per GPU for various kernels to hit 90% of asymptotic performance. Overall 3 cores per GPU is enough to hit the 90% efficiency mark for the full application.



(a) SW4Lite

(b) MiniVite

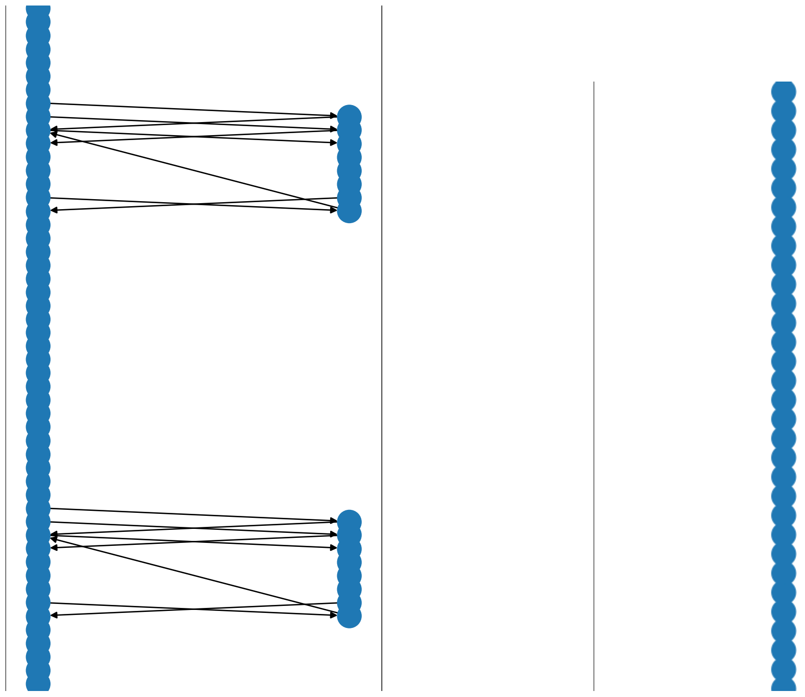
Figure 34: Execution timelines for each stream represented as Gannt charts

6.5.1 SW4Lite

SW4Lite is the most promising of all the proxy applications in terms of GPU utilization (as shown in Figure 38 with a 30% utilization compared with Quicksilver, MiniVite and Laghos that have only a 10% utilization on average). We run these experiments for both CPU and GPU versions with the gaussian-hill.rev dataset. More details about the running environment can be found in 2020 Q4 Milestone report. For the first application profile, we change the threads on the single rank to capture the scaling behavior. For the over-subscription experiments, we increased the number of MPI ranks while partitioning the GPU accordingly. We have similar setups for the Laghos, MiniVite and Quicksilver.

The CPU version takes around 1346 seconds. On the other hand, the GPU version takes 9.23 seconds. This indicates a gain of 146 times by the GPU version compared to the CPU one, suggesting noticeable advantage of using GPUs. To better understand this performance gap, we analyze individual kernel execution profile for this application. Figure 36 reports the percentage of time spent by each parallel kernel, running on a Nvidia V100 GPU. We identify that kernels rhs4_v2, addsgd_4, dpdmt_dev, and rhs4_X take the major share of the execution time for the SW4Lite application. Thus, we focus on these kernels for the analysis.

Figure 37 reports the SM efficiency for the most computationally-expensive kernels we identified from Figure



(a) SW4Lite **(b)** MiniVite

Figure 35: Partial poger generated DAGs. The complete DAGs are too big but have repeated patterns

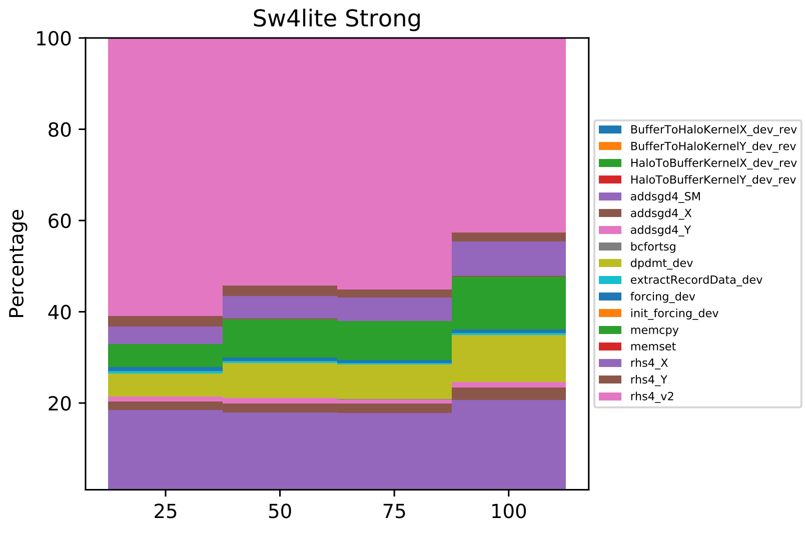


Figure 36: Breakdown of the execution time (in percentage) spent by the parallel code regions in strong scaling experiments of SW4Lite on a V100 GPU.

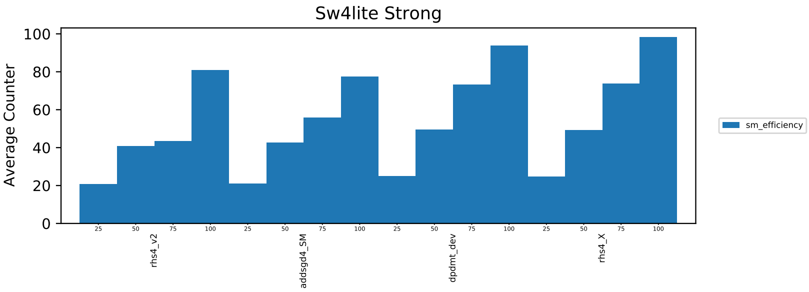


Figure 37: SW4Lite Efficiency Counters for V100

36. Bar charts in the plot for each of these kernels represent the effective SM efficiency when running strong scaling experiments with 25%, 50%, 75% and 100% of the available computational power of the accelerator enabled respectively. As can be observed from the plot, even when the application has all the SMs available (100%), the application utilizes at most around 80% of the available SMs on the accelerator. This indicates that while the application can use most of the computational resources on an accelerator, it will not use it fully.

Because of the difficulty of running weak scaling cases for SW4Lite and because of the observed better performance of the GPU version of the application, we skip doing the same exercise for the weak scaling case.

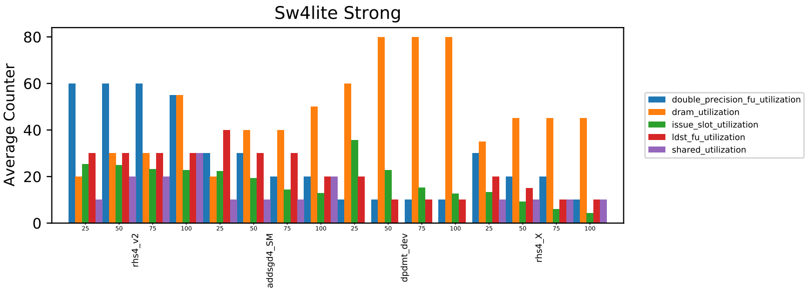


Figure 38: SW4Lite Utilization Counters for V100

Although the efficiency of the accelerator is high, when we break down the application profile by instruction family types, we note that computation (represented by floating point operations) is only 60% of the instruction mix of the most computationally-intensive kernels (Figure 38). Other kernels contain smaller number of floating point operations. In the case of the third most expensive kernel (dpdmt_dev), we observe that its execution is dominated by memory operations (as seen by the utilization of the DRAM unit).

We investigate the benefit of over-subscription on a GPU for this application. In these experiments, we assume that either the shared CPU-GPU communication link, the workers, or the memory subsystem will cause the over subscription to start giving diminishing returns, as we increase the amount of over-subscription (i.e. due to contention). We performed a strong scaling experiment using Nvidia's MPS on Newells V100. This application requires ranks in powers of three, thus we tested 1, 8, and 27 ranks on a single GPU. With these experiments, we observed a 17% improvement when running 8 MPI ranks with one GPU, and conclude a ratio of CPU vs GPU of 8:1. Thus, we can over-subscribe the accelerator by 8 without affecting (and even improving) performance. The speed up curve for this experiment can be seen in Figure 39.

6.5.2 MiniVite

MiniVite represents a special case since it is a graph proxy application (i.e., community detection proxy app) and as such is not floating point heavy. Moreover, the CPU and GPU versions differ significantly in their implementations because of the underlying architectures and the need to extract the performance out of the substrate. All experiments presented were run with a random RMAT graph of scale 21. For weak scaling experiments, we replicate this input graph.

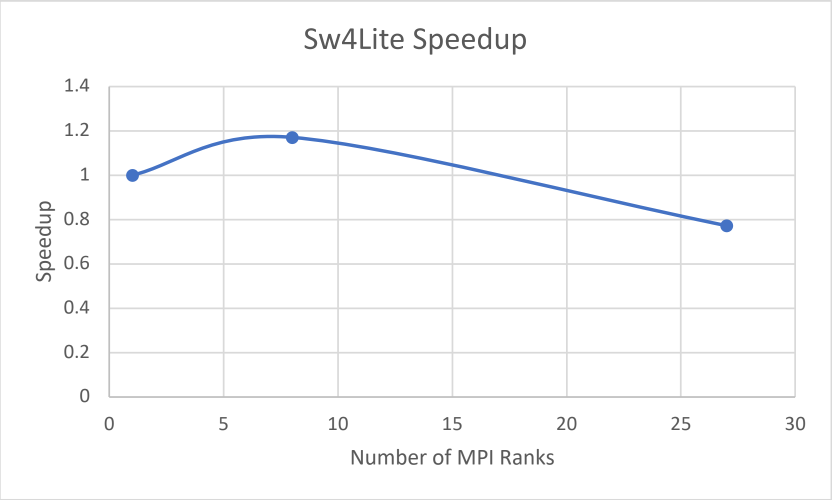
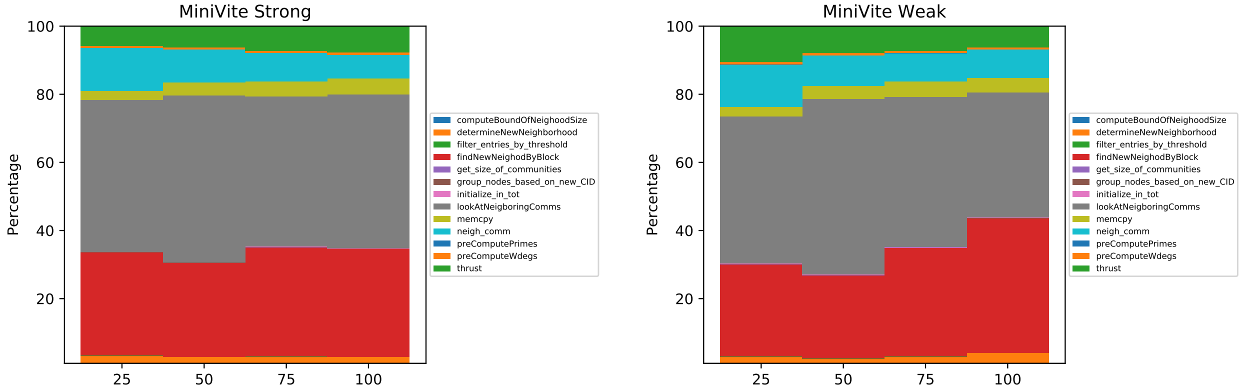


Figure 39: Normalized Speedup (w.r.t. 1 MPI rank on one GPU) of SW4Lite when running N MPI processes on one GPU (over-subscription).

Figures 40a and 40b show the breakdown of execution time (in percentage) of different CUDA kernels for MiniVite application. From these figures, we observe that the kernels findNewNeighborbyBlock, lookAtNeighboringComm, and neighcomm are the most computationally expensive kernels that takes around 85% of the total runtime. Thus, we focus on these kernels for subsequent discussion.



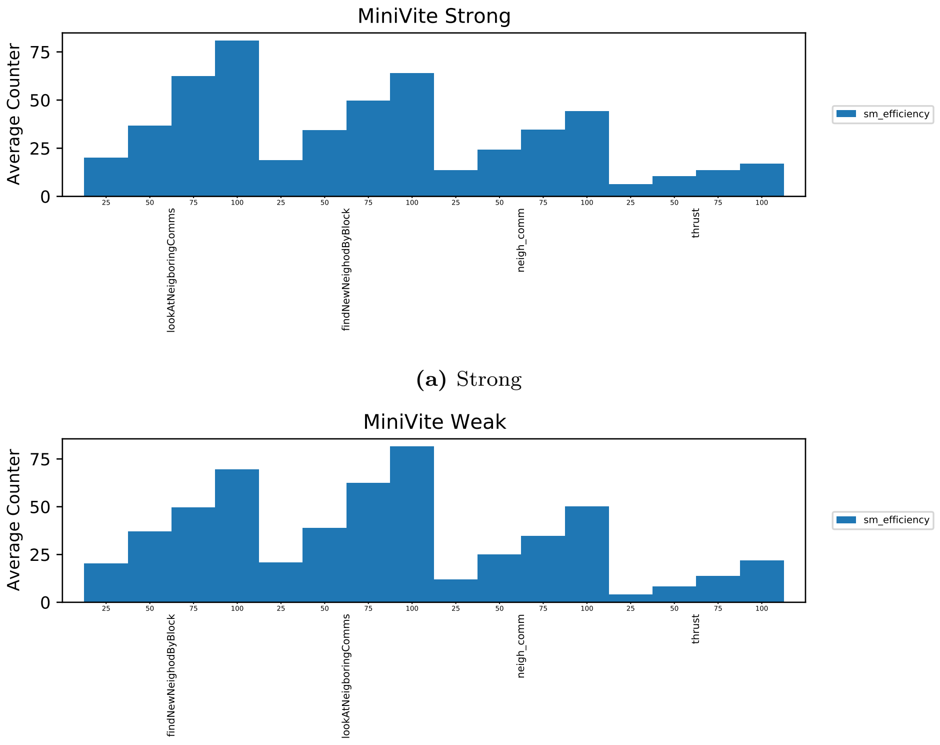
(a) Strong

(b) Weak

Figure 40: Breakdown of the execution time (in percentage) spent by the parallel code regions in strong and weak scaling experiments of MiniVite on a V100 GPU

Figure 41a and 41b report the SM efficiency achieved by these kernels when running the strong and weak scaling experiments, while varying the number of SMs available. As can be observed from these figures, MiniVite kernels achieve at most 60% to 70% of the available SM efficiency at full scale (i.e. when all the SMs on one GPU is available). It is interesting to note that even though the weak scaling experiments are performed with larger datasets, the SM efficiency does not change in a noticeable manner, compared to the strong scaling experiments.

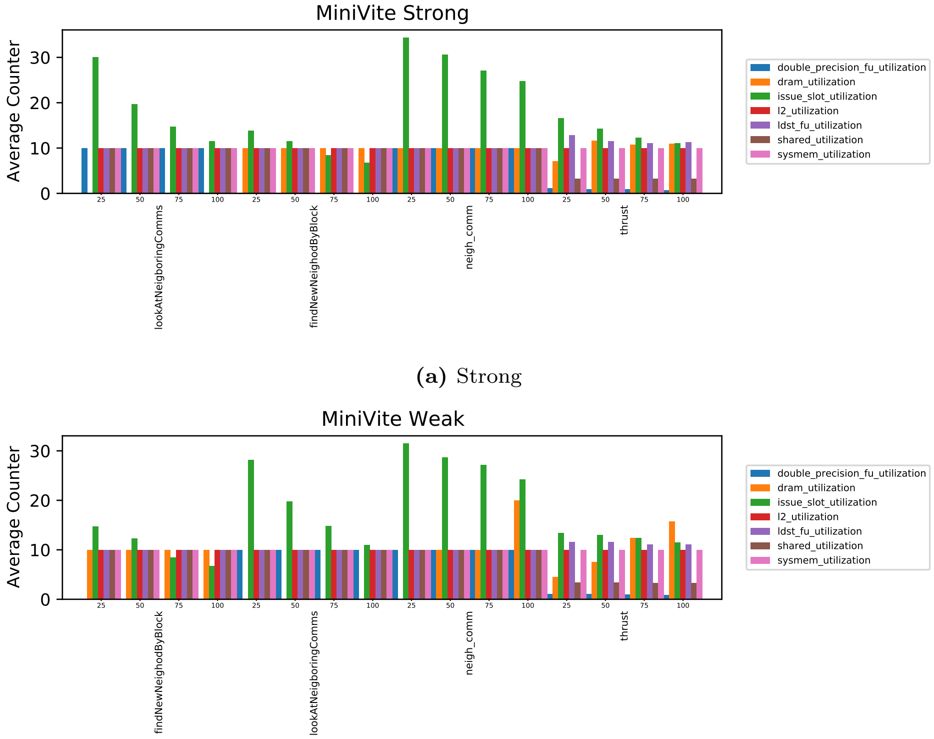
Next, we delve into analyzing the functional unit utilization by different kernels. The utilization is presented in Figures 42a and 42b. Since the computation in this proxy app is more integer-based than floating point based, issue slot utilization should be considered more significant than floating point performance. Even then, in both cases (i.e. in strong and weak scaling experiments), the major kernels utilize an average of 10% percent of the issue slot, with a maximum of around 30%, indicating that the GPU is not utilized in an efficient manner by this



(a) Strong

(b) Weak

Figure 41: MiniVite Efficiency Counters for V100



(a) Strong

(b) Weak

Figure 42: MiniVite Utilization Counters for V100

irregular workload.

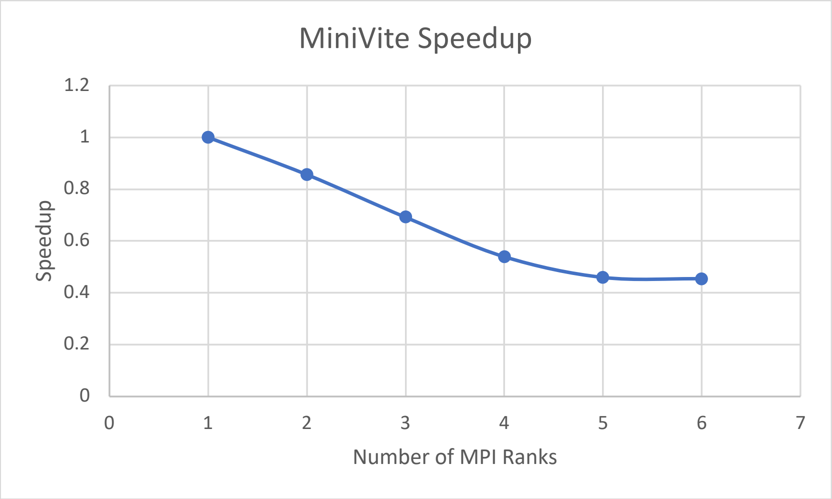
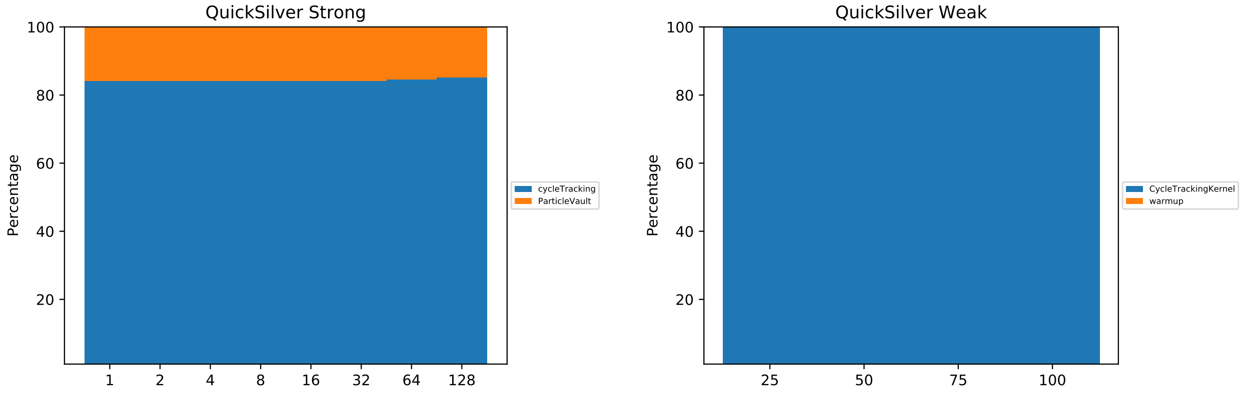


Figure 43: Normalized Speedup (w.r.t. 1 MPI rank on one GPU) of MiniVite when running N MPI processes on one GPU (over-subscription)

Similar to SW4Lite, this implementation is ill-prepared to fully utilize the GPU. Thus, we repeat the same over-subscription experiments as in SW4Lite, and report the results in Figure 43. In Figure 43, we plot speedup as a function of the number of MPI ranks on a single GPU. In this case, we are expecting the memory moved between the host and device to be the main bottleneck for achieving better performance. While theoretically, the underutilized GPU should be able to support up to 8 or 9 more MPI ranks (reaching an utilization of 100), we observe that performance degrades even running two ranks, hence we determine that the ratio of CPU to GPU is 1:1.

6.5.3 QuickSilver

The QuickSilver workflow is a communication simulation program, hence it is not as computationally-intensive as SW4Lite. However, similar to MiniVite, QuickSilver is a data-dependant application (mimicking pointer-chasing behavior). We ran this workload with a dataset of 50000 particles for both strong and weak scaling environments. The computational profile of this app is the simplest of all applications, with only one major kernel called CycleTrackingKernel as shown in Figures 44a and 44b.



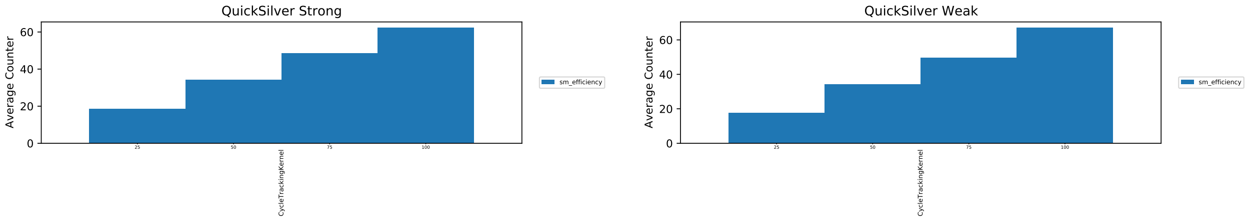
(a) Strong

(b) Weak

Figure 44: Breakdown of the execution time (in percentage) spent by the parallel code regions in strong and weak scaling experiments for QuickSilver.

As reported in Figures 45a and 45b, the kernels in QuickSilver achieve an SM efficiency of 60%-70% . Similar to MiniVite, we observe that QuickSilver does not fully take advantage of the GPU with the current dataset. When inspecting the utilization of the functional units (Figures 46a and 46b), we see that that the utilization of different functional units is at most 10% for both weak and strong scaling cases. This indicates that, this application would not benefit from an extra or bigger GPU.

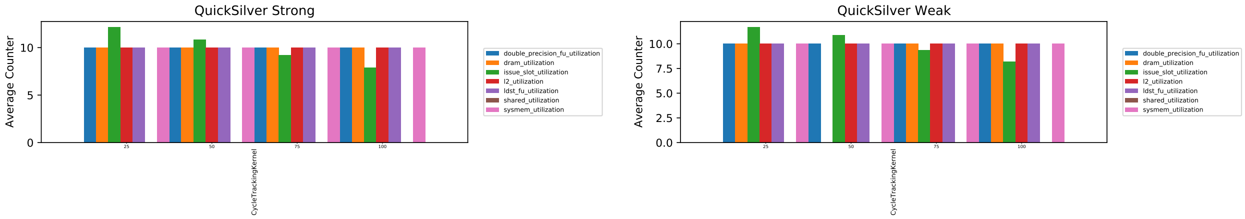
We also run an over-subscription experiment to determine the correct ratio of CPU to GPUs as done for Sw4Lite and MiniVite. In Figure 47, we present speedup relative to a single rank while scaling the number of ranks on a single Nvidia V100 GPU. While we anticipate a benefit by increasing the ranks, we observe a similar behavior to MiniVite in that we observe a slowdown for even two ranks. With this information, we conclude the CPU to GPU ratio to be 1:1.



(a) Strong

(b) Weak

Figure 45: QuickSilver Efficiency Counters for V100



(a) Strong

(b) Weak

Figure 46: QuickSilver Utilization Counters for V100

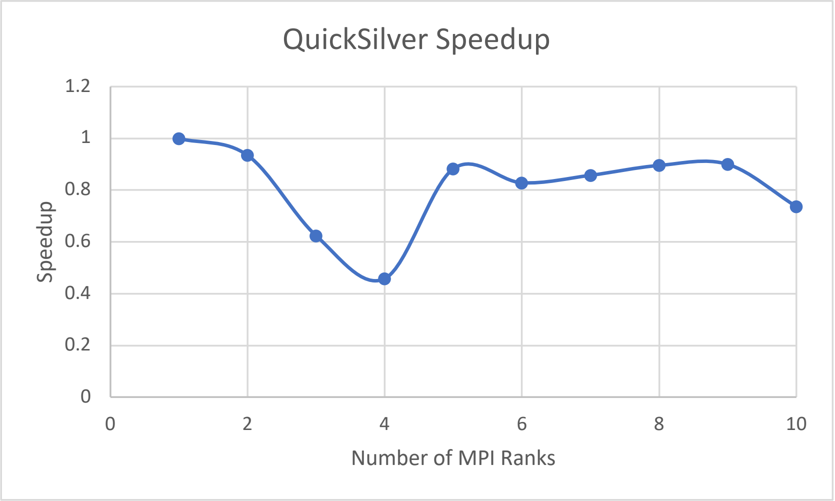


Figure 47: Normalized Speedup (w.r.t. 1 MPI rank on one GPU) of QuickSilver when running N MPI processes on one GPU (over-subscription)

Dataset	CG (H1)	CG (L2)	Force	UpdatedQuadData	Major Kernels
cube_24	15.08	0.52	0.21	3.82	18.67
cube_311	(segfaults)				
cube_522	47.49	1.59	0.66	12.29	56.33
cube_922	143.69	6.29	3.17	56.31	189.42

Table 8: Laghos CPU runtimes (in seconds)

Dataset	CG (H1)	CG (L2)	Force	UpdatedQuadData	Major Kernels
cube_24	70.51	2.06	0.35	1.03	71.87
cube_311	19.81	0.71	0.12	0.31	20.25
cube_522	150.18	7.37	0.68	1.83	152.7
cube_922	414.41	28.3	1.84	6.74	423

Table 9: Laghos GPU runtimes (in seconds)

6.5.4 Laghos

The LAGrangian High-Order Solver or Laghos is a mini-application designed to solve the time-dependent Euler equations of compressible gas dynamics which captures the basic structure of many other compressible shock hydrocodes such as the LLNL BLAST code. The code is built upon the discretization library MFEM which applies a clear separation of concerns when referring to the point-wise physics and the meshing concerns.

We ran this application to solve the Sedov Blast problem with several "cube_x" datasets (where x=24, 311, 522, 922 etc.) to stress the computational components of the applications. We compare the execution time of this application on CPU and GPU in Tables 8 and 9 for each major kernel. The most expensive kernel for both cases is the CG(1) by a large margin. An interesting part of these numbers is that the CPU version for these datasets is *faster* than the GPU version across all datasets. This may indicate that the select datasets generate insufficient work or that there exists an intrinsic behavior or requirement of the application causing a slower execution. Thus we require a further in-depth profiling of the application.

For profiling, we specify a reduced number of iterations to capture a snapshot of the workload that is sufficiently representative of the converged application with larger number of iterations. In the case of the utilization for each functional unit (presented in Table 10), we see that all the results are low or idle. These demonstrate that the overhead of offloading to an accelerator is not being amortized by the GPU execution matching the observed runtimes (i.e. the GPU's execution taking longer than the CPU's.).

Using these insights, we run the over-subscription experiment using the cube_922 dataset on a V100. Figure 48 shows the speedup relative to the number of ranks for a single GPU. We observe significant slowdown at 2 ranks and thus conclude a CPU to GPU ratio of 1:1. This workflow launches many kernels using Nvidia's Thrust library leading to heavy interactions between the CPU and GPU (as shown in Table 7). We believe this puts pressure on the communication link, prohibiting the oversubscribed ranks to further utilize the GPU.

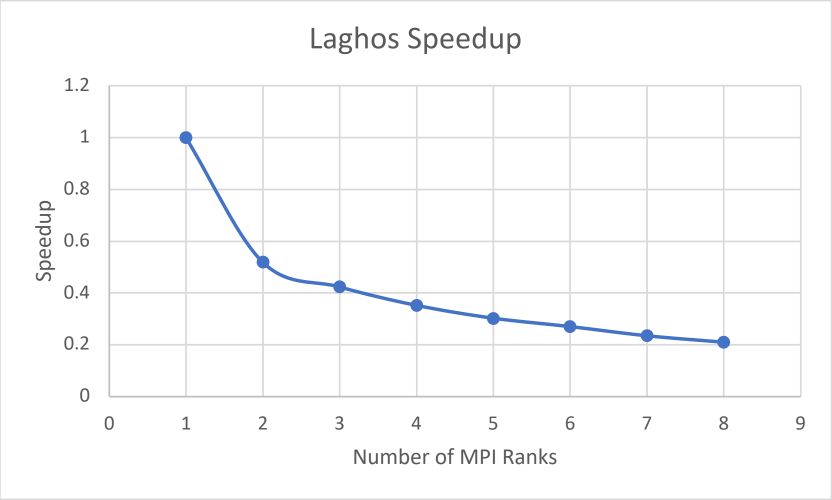


Figure 48: Normalized Speedup (w.r.t. 1 MPI rank on one GPU) of Laghos when running N MPI processes on one GPU (over-subscription).

Dataset	Counter	Average
cube_922	Double Precision FU utilization	low(2)
	Double Precision FU utilization	Low(2)
	L2 utilization	Low(1)
	Issue Slot utilization	13.46%
	Shared Memory utilization	Idle(0)
	Texture Memory utilization	Low(1)
	System Memory utilization	Low(1)
	load store FU utilization	Low(1)
	sm efficiency	87.79%

Table 10: NVPROF results for Laghos Utilization

7. PENALTIES FOR CPU-GPU DISAGGREGATION

For nearly two decades, parallel HPC file system appliances have been deployed in supercomputing environments in lieu of a node-local storage as the former provides centers with scalable bandwidth, much higher utilization, and bandwidth/storage QoS and apportionment. Given accelerators are PCIe-attached devices that are not utilized in every application or continually in every application, one must wonder if a similar approach to disaggregating nodes and pooling resources would be appropriate for HPC simulations. To that end, we model the performance of our HPC application suite under the assumption that GPUs (including their HBM memory) are now remote devices accessed over the global network. The local CPU must now dispatch kernels to the remote GPUs. In doing so, we assume the CUDA kernel launch time is increased by 10× and the D2H and H2D bandwidths have decreased by 10×. These numbers may be a bit extreme but do account for either the long-tail observed in HPC networks or the expected future scaling discrepancy between GPU performance and network bandwidth/latency.

7.1 AMREX

In Section 5.1 we observed that not all AMReX applications can benefit from tighter CPU-GPU integration. This motivates a study on whether these codes can be ported on a disaggregated system with acceptable performance penalty. Table 11 summarizes moving costs in AMReX applications that are likely to be amplified. We observe low overhead in Castro (mostly due to the modest amount of parallelism offloaded to the GPU), explaining for the very low penalty. For other applications, the slowdown ranges from 2.99× to 5.79×.

Application	Kernel Calls	Launch overhead	UVM	cudaMemcpy	Penalty
PeleC	1.8e6	0.03%	53.0%	0.02%	5.79×
Castro	1.5e4	0.20%	0.07%	0.26%	1.05×
MFiX	7.1e6	18.00%	0.10%	3.98%	2.99×
WarpX	9.8e6	29.40%	0.50%	1.11%	3.79×

Table 11: Performance penalty can be driven by either data movement cost (PeleC) or fine-grain kernel launches (MFiX and WarpX). Castro doesn't use GPU extensively, thus using a remote or local GPU will not have much impact on the overall performance.

7.2 BERKELEYGW

The impacts of CPU-GPU disaggregation on BerkeleyGW were estimated using the same methods (see Section 4.2) and the same data (see Section 5.2) described for CPU-GPU integration. The scaling factors for t_{CGdata} and $t_{CGcontrol}$ were adjusted to reflect *poorer* bandwidth and latencies on a disaggregated system. Setting $\lambda_{CGdata} = \lambda_{CGcontrol} = 0.1$ leads to a 5.7× performance penalty for dissaggregation for Epsilon, and a 5.8× penalty for Sigma.

7.3 GTCP

For GTCP, Leveraging the model in Section 4.2, t_{CGdata} has a greater impact on performance compared with $t_{CGcontrol}$. Using the parameters in Table 5, disaggregating CPU from attached CPU significantly impact the performance due to the increased cost for data movement. For instance, a reduction in bandwidth by 10× due to disaggregation, results in 2.44× increase in the total application execution time. The latency component impact on performance is less profound. For instance, increasing the latency by 100×, while maintaining bandwidth, increases the execution time by 18%.

7.4 SPTRSV

The impacts of CPU-GPU disaggregation on SpTRSV are mainly relying on the NVSHMEM performance. According to the NVSHMEM installation Guide [20], the requirement of using NVSHMEM is that all GPUs must be P2P-connected via NVLink/PCIe or via GPUDirect RDMA over InfiniBand/RoCE with a Mellanox adapter (CX-4 or later). Therefore, we use the NVSHMEM performance via NVLINK (red line in Figure 18a) and IB (black line in Figure 18a) on Summit to estimate the poorer bandwidth for CPU-GPU disaggregation. We then lower the NVSHMEM on-node performance by 4×. Thus, with a 4× lower bandwidth, the overall SpTRSV performance decreases by 2.5× for matrix S1 and 1.1× for matrix Li using six GPUs via IB compared to six GPUs via NVLINK.

7.5 HACC, LAMMPS, AND OPENMC

In order to disaggregate tasks to a GPU located in a remote host node, data needs to move to the main memory of the remote node first, and then move to the GPU memory from the remote host's main memory, thus both the data transfer latency increases significantly. This is also true for control commands such as kernel invocation and synchronization. While interconnects between nodes may have higher bandwidth, the bandwidth for disaggregation may be limited by the PCIe bandwidth between host and GPU, or vice versa. We need to choose the lowest bandwidth number in a system. We estimate the CPU-GPU disaggregation penalties using the same method that is used to estimate the CPU-GPU integration benefits in Section 5.5 using the AgoChart simulator.

7.6 ESTIMATION OF THE DIS-AGGREGATION PENALTY WITH SSI FOR SW4LITE, MINIVITE, QUICKSILVER, AND LAGHOS

As described in Section 5.6, we use the simulation infrastructure to calculate the integration penalty using the formula described in that section (i.e., $IntegrationPenalty = \omega/\delta$). Given our approximations of the current lane lengths, the maximum dis-aggregation will be larger than our aggregation since our the current technology (i.e., 3 propagation delay) is much closer to our proposed minimum lane length (i.e., 0 propagation delay) relative to the maximum lane length (i.e., 50 propagation delay).

7.6.1 SW4Lite

We report the penalties of moving the accelerator apart in Figure 24 (refer to Section 5.6 for the figure), The penalties in these cases ranges from 1.13x for NVLink2 and 1.40x for PCIe. This behavior is expected because the latency is the key factor since we are not saturating the bandwidth.

7.6.2 MiniVite

As presented in Figure 25 (refer to Section 5.6 for the figure), the penalties of MiniVite are more interesting due to its larger data movement across the links. For these cases, we see a penalty of 5.34x for NVLink2 and 8.6x for PCIe when dis-aggregating the accelerators. We note the penalty is much larger compared to the aggregation benefit, due to the fact that our estimation for the current technology is closer to our minimum lane length. We further note the penalty across technologies differs significantly as we assume the transmission delay is fixed.

7.6.3 QuickSilver

Using the results presented in Figure 26 (refer to Section 5.6 for the figure), we calculate the dis-aggregation penalties for QuickSilver to be around 1.15x for NVLink2 and 1.41x for PCIe. This result is similar to our findings for Sw4Lite, as the number of kernels and the amount of data transmitted is minimal.

7.6.4 Laghos

Using the results presented in Figure 27 (refer to Section 5.6 for the figure), we calculate the dis-aggregation penalties for Laghos to be around 12.5x for NVLink2 and PCIe4 due to the larger number of kernel calls (instead of a large number of transmitted bytes).

8. BENEFITS OF TIGHTER GPU-GPU INTEGRATION

Historically, MPI-based parallel computing envisions a sea of CPUs (cores) interconnected by a scalable network. However, today, the reality is that networks are hierarchical including on-chip and on-node networks that interconnect cores on a socket (NoC) and sockets on a node (e.g. Intel UPI). Moreover, there is substantial tapering in the global network from chassis, to racks to systems. Programmers are thus incentivized to map processes and exploit on-node shared memory communication to avoid network bottlenecks. GPU manufactures have embraced such an approach in recent years (NVLINK interconnected GPUs) likely due to the need to exploit model parallelism in Deep Learning applications without abandoning the standard PCIe CPU-GPU interface.

Assuming CUDA-aware MPI can likely exploit NVLINK on a node, in this section, we examine how having NVLINK play a slightly larger role in the overall network (i.e. more NVLINK-interconnected GPUs on a node and fewer nodes) will affect HPC application performance. Concurrently, we can increase NVLINK bisection bandwidth by increasing NVLINK link bandwidth or increasing the radix of the network.

We acknowledge that such designs will need to increase the number of CPUs on a node (to preserve the CPU:GPU ratio). Concurrently, unless noted, we assume NIC bandwidth per node will remain fixed. This analysis assumes GPU-GPU integration is implemented independent of any other integration/disaggregation technology.

8.1 AMREX

Adaptive mesh refinement workloads can have diverse communication patterns since data not only moves between partitions of a single mesh but also across meshes (i.e. between coarse- and fine-grain mesh levels). This diversity is more pronounced if the input problem requires frequent regridding so that serious load imbalance can be avoided. In this section we study the communication behaviors of four AMReX applications and analyze the benefit of having more GPUs on the same node (tighter GPU-GPU integration). We use IPM to instrument MPI invocations and collect communication traces. We split MPI communication activities into point to point (P2P) and collective, on-node and off-node communication as shown in Table 12. Communication time with respect to the total execution time is described by the %Comm column. %P2P is the percentage of point-to-point communication volume while the rest is for collective operations. For point-2-point messages, %off-node denotes the part of data volume that moves between nodes while local send/recv data volume is 100%-%off-node.

Application	%Comm	%P2P	%Off-node	Speedup on 16-GPU node
Castro	7.8%	36%	12%	1.00 ×
MFIX	75%	99%	29%	1.23 ×
PeleC	10%	80%	25%	1.02 ×
WarpX	40%	98%	22%	1.09 ×

Table 12: Despite using the same AMR framework, we observe different communication behaviors on the four applications. As a result, the performance benefit of a tighter GPU-GPU integration will vary from no benefit to around 1.2×

Currently, on Cori-GPU, 8 GPUs on the same node are wired into a hyper-cube where communication between 2 arbitrary GPUs is secured with a 25GB/s bandwidth for each direction. In contrast, all off-node messages have to share the same path with limited bandwidth. On Cori-GPU, 2 nodes are connected via 2 NICs, each supporting 16GB/s uni-directional bandwidth (i.e. 32GB/s total). When this path is shared between multiple source and sink processes, the aggregate bandwidth can be a bit lower. We use the OSU multi-pair bandwidth test and realized 24GB/s. A fatter node with more GPUs and direct links among GPUs will speed up P2P off-node messages by 25× 8/24= 8.3×. Collective operations, however, will not benefit much from such a node architecture, since one can use a custom algorithm so that most communication can be localized (i.e. the shared path between two nodes is not the bottleneck).

Figure 49 explains why some applications can benefit from tighter GPU-GPU integration while others will not. Here we run 4 AMReX applications on 16 GPUs (2 nodes). Since we use linear mapping, the first 8 MPI processes are mapped to the first node while the other 8 run on the second node. With this setup, on-node communication can be represented by cells in the bottom left (P0:P7, P0:P7) and upper right (P8:P15, P8:P15) areas while upper left (P0:P7, P8:P15) and bottom right (P8:P15, P0:P7) areas show off-node communication. It can be seen that MFIX moves much more data across nodes (e.g. P0-P12, P0-P14, P1-P13, P1-P15, P2-P8, P2-P12, P3-P10, P3-P11, etc). As we have already seen in Table 12, MFIX communication costs 75% of the total execution time and 99% of communication is P2P. This explains why tighter GPU-GPU integration is beneficial to MFIX (1.23× speedup). WarpX has notable off-node communication, but %Comm is only 40%. Thus, the expected speedup is only 1.09×. Castro, however, does not move much data across nodes. In addition, %Comm and %P2P of Castro are also low. PeleC has more off-node communication than Castro, but %Comm is also very low. As a result, we expect no performance benefit for Castro and very modest speedup for PeleC if running these applications on a fat node with tighter GPU integration.

8.2 BERKELEYGW

We considered two potential directions for GPU-GPU itegration for BerkeleyGW: a) improved intra-node GPU bandwidth, and b) increasing the number of GPUs in a node. The current version of BerkeleyGW uses only one GPU per process, and all communication between GPUs occurs through MPI. Thus, our analysis is based on the MPI profiling through the IPM [26] library.

The BerkeleyGW-Epsilon problem was profiled using two Cori-GPU nodes, with a total of 16 MPI processes, one V100 GPU per process and five OpenMP CPU threads per process. This configuration uses the minimum number of nodes to satisfy the job memory constraints, and uses the full computational resources of the node. The IPM profile showed that 8.8% of the walltime was spent in MPI; specifically 5.8% was spent in large-payload collective functions (`MPI_Bcast`, `MPI_Reduce` and `MPI_Allreduce`), 1.6% in `MPI_Barrier`, 1.4% in large-payload point-to-point functions, plus small contributions from various other functions. The presence large-messages in the MPI profile points to utility of increased GPU-GPU bandwidth, but its precise value depends on the algorithm used to implement the collective functions. For concreteness, we assume a binary tree algorithm. For this two-node job, at least one of the four-stages in the communication tree must cross node boundaries and would not be accelerated by a faster intra-node network. Suppose the on-node network was improved by a factor of 2×, then the runtime would decrease to $(100 - 8.8) + 5.8(\frac{1}{4} + \frac{3}{4}/2) + 1.6 + 1.4/2 = 97.1\%$ of the current value, a mere 2.7% performance boost On the other hand, if the node was expanded to include 16 GPUs, then all four stages could be accelerated, providing 1.2% additional peformance.

The BerkeleyGW-Sigma problem was profiled using one Cori-GPU node with 8 MPI process, one V100 GPU per process and OpenMP CPU threads per process. Like with Epsilon tests, this configuration uses the minimum number of nodes to satisfy the job memory constraints, and uses the full computational resources of the node. Sigma's MPI profile is even more heavily dominated by `MPI_Bcast`; 22.8% of the walltime was spent in MPI, including 16% in `MPI_Bcast` and 5% in `MPI_Wait`. At this scale, there is no off-node communication, so assumptions about the `MPI_Bcast` implementation are not necessary; and all bandwidth-limited MPI functions would be comparably accelerated. The `MPI_Bcast` calls have a uniform signature, with 250 kB sent from rank-0 to all others, a size that is clearly bandwidth-constrained on this architecture. The `MPI_Wait` calls correspond to bandwidth-constrained non-blocking point-to-point calls (`Isend` and `Irecv`). If we again suppose a 2× bandwidth between GPUs within the node, Sigma performance would increase by 12%.

Practical BerkeleyGW simulations (for both Epsilon and Sigma) are often larger than the one- and two-node jobs analyzed here. (Some BerkeleyGW jobs have used up to 27,648 V100 GPUs on 4,608 Summit nodes [1].) As the node-count grows, a larger fraction of the MPI_Bast traffic must traverse the inter-node network, and the fraction of traffic that could be accelerated by an enhanced on-node interconnect between GPUs decreases. The preceding approach is therefore likely to overestimate the speedup from tighter GPU-GPU integration. The magnitude of this effect would depend on both the node count selected and the degree of bandwidth tapering.

8.3 GTC-P

GTCP does not have any explicit data movement management between GPUs sharing the node. Each MPI rank leverage a single GPU to accelerate critical phases of the computation. The computation pattern is a simple ring (or two rings in the case of poloidal decomposition). The GPU integration potentially reduces the GPU-GPU communication cost, which is mainly BW limited for the dominant messages in GTC-P. Improving the communication cost through tight integration will reduce the communication time. At a single node scale, 4× improvement in the bandwidth between GPUs due to integration would result in roughly 1.17× improvement in the overall performance. This estimate assumes that the core per GPU ratio and associated BW to memory will remain the same as we integrate more GPUs. While scaling, we expect a slightly lower benefit for tighter integration of GPU as the off-node communication becomes more critical to performance. However, our earlier experience shows that the ring communication pattern allows efficient scaling to a large number of nodes.

8.4 SPTRSV

The SpTRSV performance of GPU-GPU integration is estimated by supposing a super GPU with N times the bandwidth while the NVSHMEM performance remains the same. First of all, let's recall how we estimated the memory bandwidth of each super node in the model. Basically, the memory bandwidth scales with the number of DAG nodes in the same DAG level until the aggregate memory bandwidth reaches the peak. On V100, the empirical HBM bandwidth is 828 GB/s [32]. According to the white paper of NVIDIA Tesla V100 accelerator [27], the maximum number of thread blocks per V100 is $\frac{80SMs \cdot 64warps}{8warps} = 640$, where 8 warps per thread block is based on our design. Therefore, we set the lower bound of memory bandwidth per thread block to 1.3 GB/s. However, the number of active thread blocks can be much smaller than the maximum of 640 due to either data dependencies or hardware limit. That is to say, in some cases, the memory bandwidth of each super node (thread block) can be larger than 1.3 GB/s. Let's consider the question of how many thread blocks can leverage a full bandwidth of a SM with dependencies. Ideally, the smallest number is two. One thread block spin waits the dependency and the other one can perform other independent computation work. Thus, the bandwidth per thread block is $\frac{828GB/s}{80SMs \cdot 2warps} = 5.2$ GB/s, and the total number of super node that can achieve the peak aggregate memory bandwidth is 160. Correspondingly, the upper and lower bound of memory bandwidth of each super node is 5.2 GB/s and 1.3 GB/s. Thus, GPU-GPU integration makes upper bound of memory bandwidth N times larger.

We can refer the yellow bar (computation time) in Figure 18b to estimate the GPU-GPU integration for SpTRSV. We can get up to 1.2× speedup for Li matrix and up to 16.6× speedup for S1 matrix when integrating twelve GPUs on one node compared to two nodes of six GPUs.

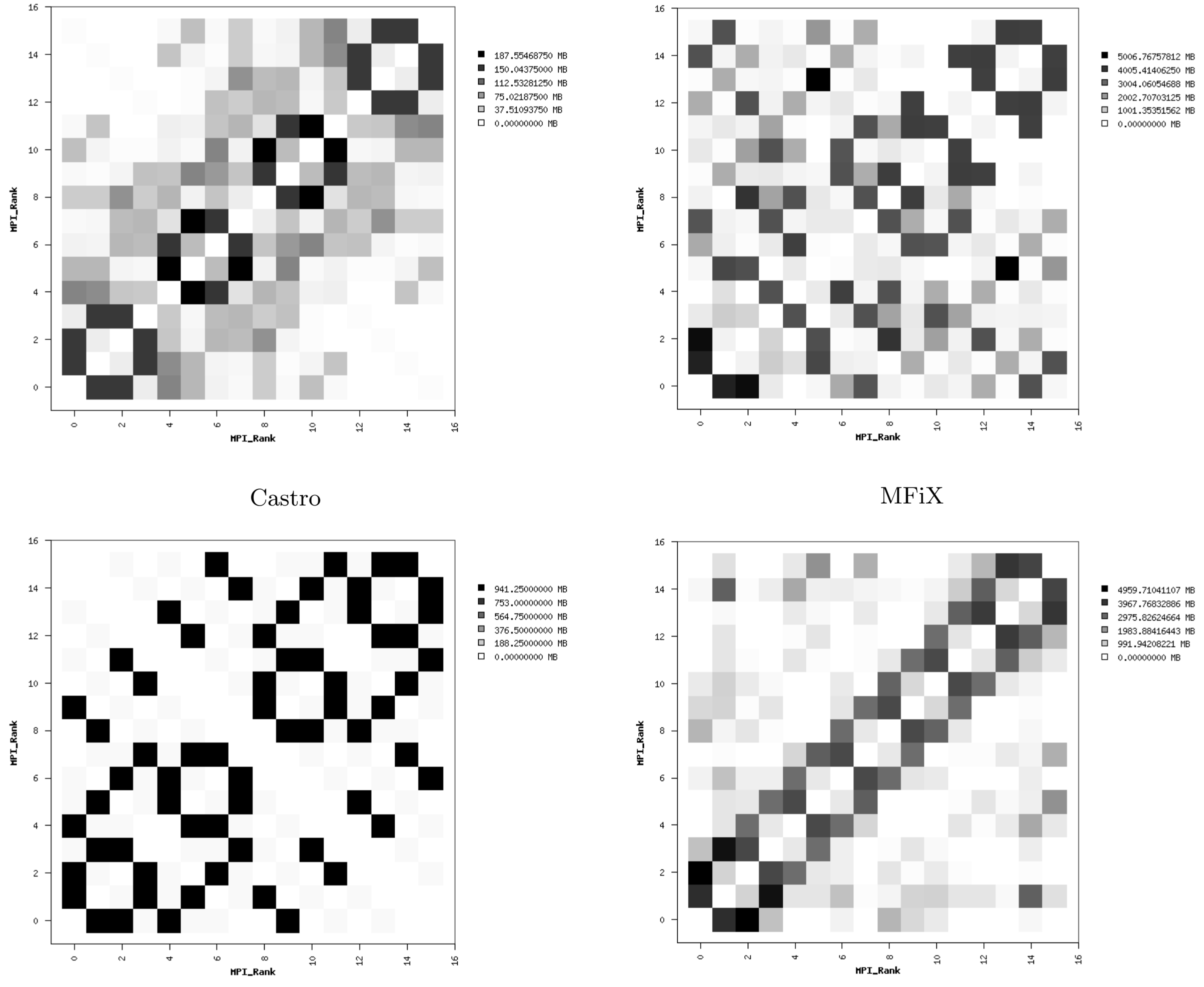
8.5 HACC, LAMMPS, AND OPENMC

If multiple GPUs are connected directly via dedicated interconnect links, we can view such tightly integrated multiple GPUs as a single larger GPU as the communication latency becomes small. An example of tightly integrated multiple GPUs is multiple-die GPUs connected via in-package high density interconnect such as Intel embedded multi-die interconnect bridge (EMIB). In such an optimal hardware configuration, the host CPU can combine multiple parallel requests to multiple GPUs into a single series of requests, which can, in turn, reduce communication latency and congestion on an interconnect between CPUs and GPUs such as PCIe, CXL.

First, the AgoChart simulator is capable of simulating loosely integrated multiple GPUs, which are multiple GPUs that share a single interconnect such as PCIe. In other words, it simulates congestion in communication traffic in PCIe using a shared resource model. We use the results on loosely integrated multiple GPUs as reference baselines. To simulate the performance of tightly integrated multiple GPUs, we simply increase the total number of execution units per GPU and run the simulator. We calculate the GPU-GPU performance benefit by dividing the performance of tightly integrated multiple GPUs by one of the loosely integrated GPUs.

Using the same set of tracing data on our applications (HACC, LAMMPS, OpenMC), which are used in both the CPU-GPU integration and the CPU-GPU disaggregation, we run the AgoChart simulator to estimate the GPU-GPU performance benefit, varying the number of GPUs (1, 2, 3, and 4 GPUs) and the data transfer and execution latency (1, 5, 10, and 50 µsec). To simplify graphs, we set both the data transfer latency and the execution latency the same number while the tool itself allows us to set these numbers independently. Figure 50, 51, 52 show that the performance projection results on HACC, LAMMPS, and OpenMC, respectively. The numbers in the GPU-GPU integration column in Table 13 are calculated by dividing the projected performance gain on the tightly integrated GPUs by that on the loosely integrated GPUs on three-GPU configuration with 10 µsec latency. We choose three GPUs here since the Aurora node has three GPU per CPU socket.

OpenMC requests the kernel invocations two to three orders of magnitude more times than the other two applications and each kernel spends a relatively shorter period of time, which stresses the shared PCIe link, creating traffic congestion, on loosely integrated multiple GPUs. On the other hand, Tightly integrated multiple GPUs allow hosts to combine requests together, which can mitigate the congestion and in turn result in higher GPU-GPU integration benefits if applications with frequent, short-duration kernel invacations.



Castro

MFiX

PeleC

WarpX

Figure 49: Heat maps showing the communication topology when running AMReX applications on 2 nodes, each having 8 GPUs. Castro communication is almost localized within a node (dark cells in the bottom left and upper right areas) and very sparse across nodes (upper left and bottom right), whereas MFIX and WarpX move significant amount of data across nodes. Although PeleC also has notable off-node communication, the impact is modest due to the low communication cost relative to the total execution time.

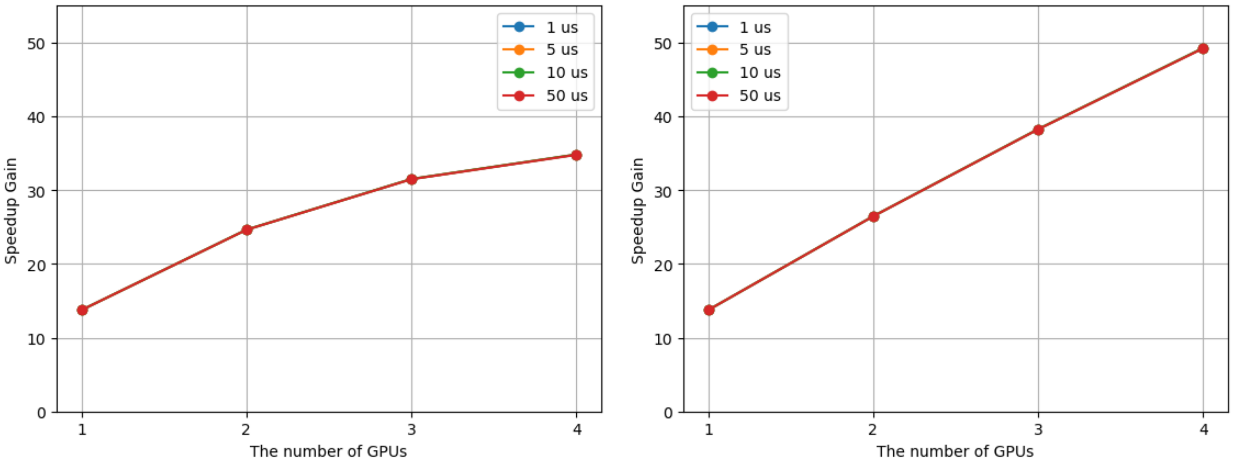


Figure 50: Performance projection of HACC on loosely-integrated multiple GPUs (left) and tightly-integrated multiple GPUs (right)

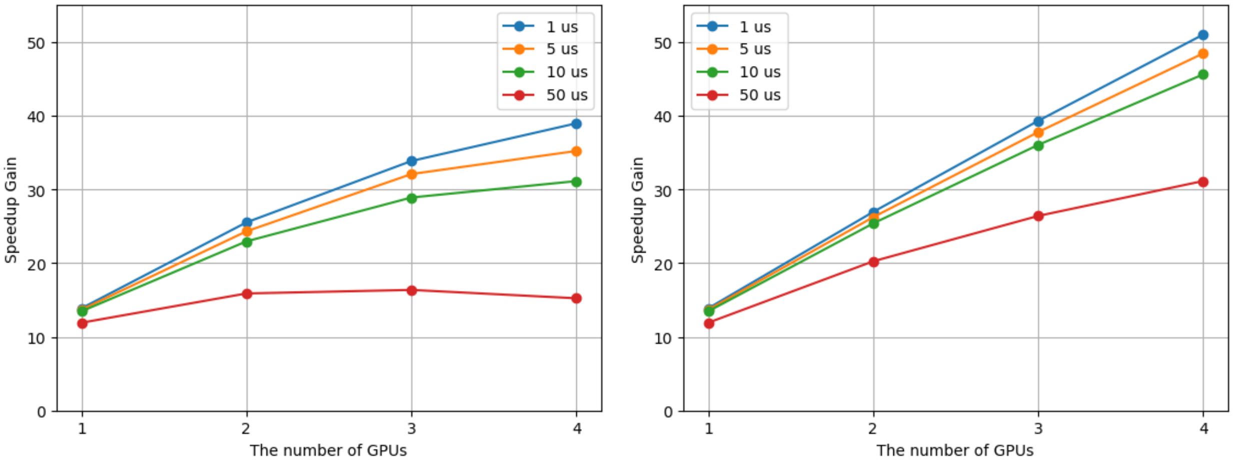


Figure 51: Performance projection of LAMMPS on loosely-integrated multiple GPUs (left) and tightly-integrated multiple GPUs (right)

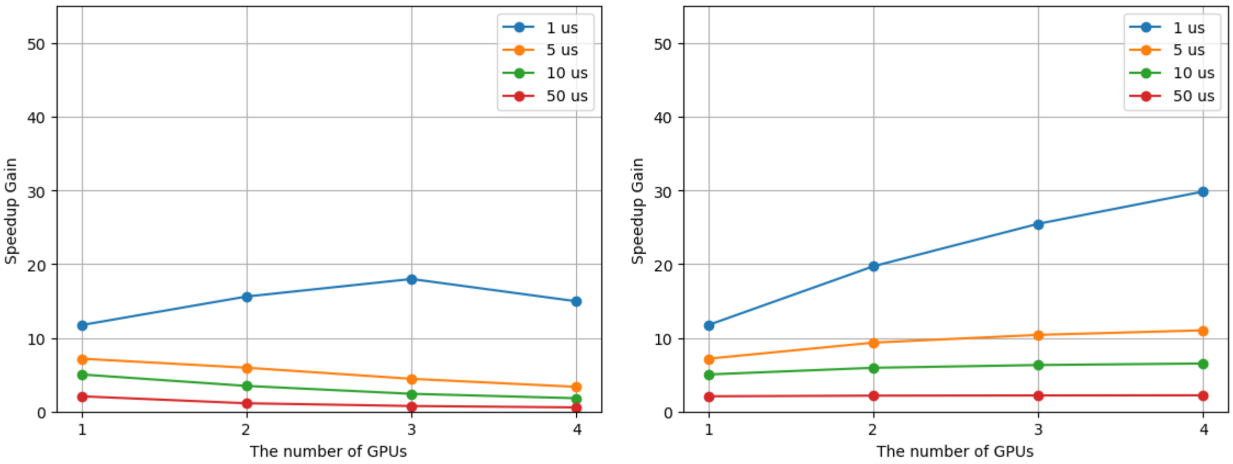


Figure 52: Performance projection of OpenMC on loosely-integrated multiple GPUs (left) and tightly-integrated multiple GPUs (right)

9. PERFORMANCE IMPLICATIONS OF UNIFIED MEMORY

Unified memory is a technology that unifies the host (i.e., CPU) and device (e.g., GPU and FPGA) memory spaces virtually and provides a single pointer for the whole system memory. Before the invention of unified memory, the device (e.g., GPU) programming required explicit data movement requests of the host and device memory objects between host and device. With unified memory, host and device can operate on the same data object without specifying the data movement between host and device, which greatly reduces the complexities of managing memory in device programming. However, unified memory might not be helpful for reducing the data movement overhead as it is controlled by the runtime and the physical locations of data also determines if actual data movement will happen. In CUDA, there are mechanisms that users can suggest the data locations (i.e., DDR-only on CPU's memory or HBM-only on GPU's memory). We evaluate the actual performance of suggesting DDR-only and HBM-only for Quicksilver. For FPGA, the CXL technology enables a cache-coherent unified memory space between host and device. We also estimate its performance for ORNLapp by using our CLX model.

9.1 QUICKSILVER — PERFORMANCE WITH UNIFIED PHYSICAL MEMORY IN CPU + GPU ARCHITECTURES

QuickSilver is a proxy application for Mercury that solves particle transport problems using the Monte Carlo Method. In QuickSilver, particles interact with matter in a variety of reactions (e.g., collision and fission) and its workloads usually contain a large amount of different reactions, which present distinct memory access patterns. To study how QuickSilver and its workloads (NoCollision, NoFission, CORAL and CTS2) perform on CPU+GPU architectures with the employment of physically unified memory, we measure its performance on a Summit node (i.e., IBM Power 9 integrated with Nvidia V100s via NVLink) by using FOM (monte carlo segments per second). Byfl 1.8.0 is used to analyze the memory access patterns (i.e., regular and irregular memory operations) of QuickSilver.

Figure 53 shows that QuickSilver is very memory-intensive (i.e., arithmetic intensity is less than 0.02 flops/byte). QuickSilver uses thin-thread model to implement its Monte Carlo Particle Transport kernel. Thin-thread model features low data races and fast data communication through shared atomic operations, which help exploit GPU performance. The memory access pattern of QuickSilver is relatively regular its QuickSilver workloads have more than 35% memory accesses that are regular.

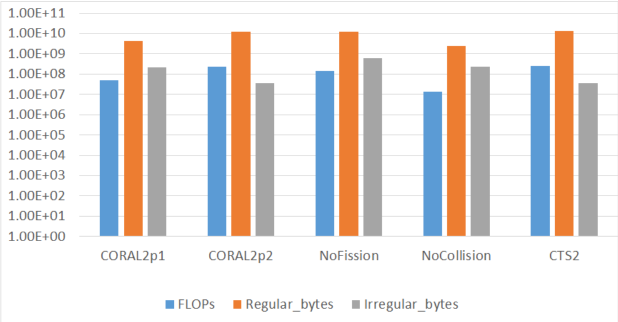


Figure 53: QucikSilver's memory access patterns

Figure 54 shows the memory bandwidth of regular and irregular memory access patterns. We use the GPU stream and pointer-chasing benchmarks and switch between suggesting CPU's DDR4 and GPU's HBM2 as the data locations to measure the memory bandwidth by using CUDA's unified virtual memory. It can be seen from the results that all five different QuickSilver workloads perform much better for HBM-only than DDR-only as DDR4 has lower memory bandwidth than HBM2 and its access also covers the overhead of direct access from NVLink. DDR-only can only achieve only 0.3–0.7× of the HBM-only performance for the five workloads and the CTS2 workload has the lowest DDR-only speedup. This could be because the CTS2 workload has the highest ratio of regular bytes to irregular bytes (357 in Figure 53), which also indicate that DDR-only might potentially benefit programs with lots of irregular memory operations.

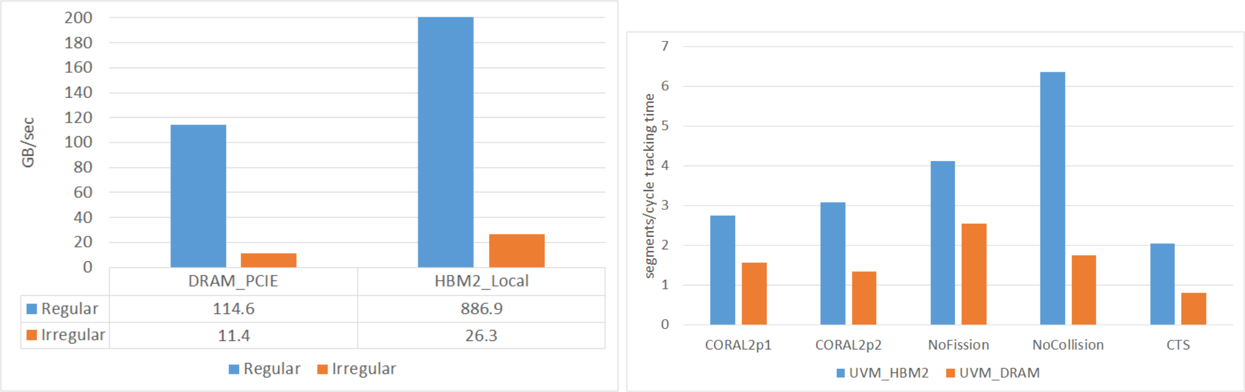


Figure 54: Evaluation of QuickSilver on V100 GPU

9.2 ORNLAPP—PERFORMANCE IMPLICATIONS OF CXL UNIFIED VIRTUAL MEMORY

The simple models discussed in Section 4.8.4 are used to gain an understanding of expected CXL performance compared to traditional PCIe transfers. Figure 55 shows the expected data transfer time as calculated by (9), (10), and (11) for PCIe, CXL model 1, and CXL model 2 respectively. The CXL model 2 plot shows the performance for both a CXL penalty of 60% and 90%.

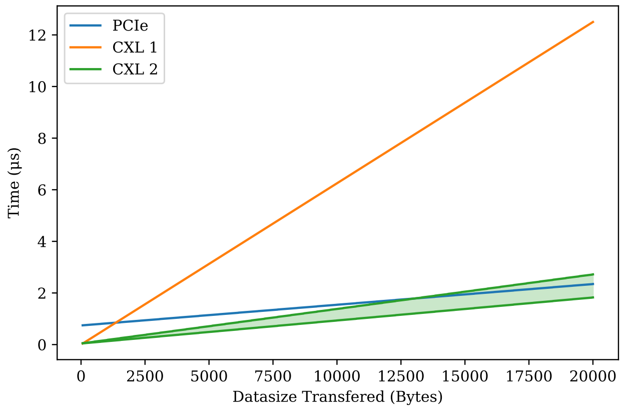


Figure 55: CXL Model: Single Transfer time for various data sizes.

As expected, the time taken for the transfer is roughly linear to the datasize being transferred. Small datasizes for the PCIe and CXL 2 models are slightly non-linear since the expected PCIe bandwidth is also based on the datasize being transferred. As shown in the graph, the CXL 1 model outperforms PCIe for small transfers of less than 1.3 KiB and the CXL 2 model outperforms PCIe for transfers less than 12.7 KiB when using a 60% penalty. Although the CXL models are rough approximations of expected CXL performance, they both show the expectation that small transfers will be faster using CXL since CXL has been designed for low-latency, and that large transfers will still be more efficient using cxl.io or PCIe to perform the bulk transfer.

Figure 56 shows the relative transfer time of the CXL models compared to PCIe, where y-values below 1 indicate better performance than PCIe. Just like before, this figure shows that for small chunk sizes the relative transfer time is less than 1. For large data sizes the relative transfer size converges to the ratio of the PCIe bandwidth. For CXL model 2 this ratio is one over the CXL penalty parameter. This penalty is currently set to be a pessimistic 60% of PCIe performance as discussed in Section 4.8.4. If we use the model to get the relative performance of CXL for transferring sizes of 16 MiB, the relative communication time of CXL Model 2 with a penalty factor of 60% to PCIe is 1.66578. One over the relative time is 0.6003 which is close to the convergence value of 0.6. Therefore the penalty of using CXL is equal to the CXL penalty parameter, which as reported from [6] is expected to be between 60–90%. Thus the expected penalty for using CXL for a cache coherent unified

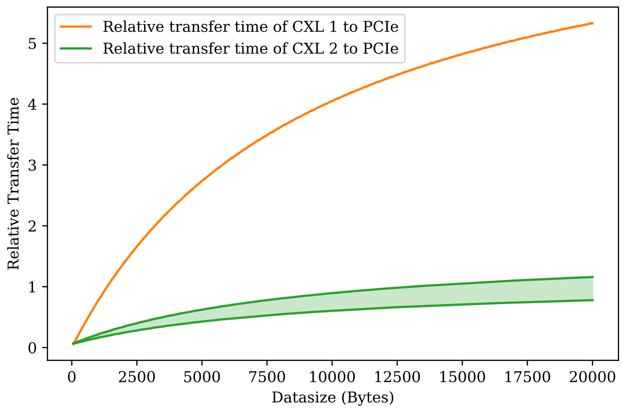


Figure 56: CXL Model: Relative transfer time of CXL and PCIe.

memory space is 0.6× to 0.9× as shown in Table 13.

10. BENEFITS OF TIGHTER FPGA-GPU INTEGRATION

In order to establish a baseline for the potential benefits of tighter GPU-FPGA integration, we use the aforementioned application that combines decompression on the FPGA with numerical integration on the GPU and measure the kernel run times on each device, as well as the total run time of the entire application. Currently, the application is structured such that there is a host to FPGA (H2F) memory transfer, followed by a execution of the FPGA kernel (F). Then the data is copied to the GPU via an FPGA to host transfer (F2H) followed by a host to GPU transfer (H2G). Next the GPU kernel is executed (G) then the data is copied back to the host (G2H). These steps are shown in Figure 57a. This execution flow is very common for heterogeneous apps with loose integration.

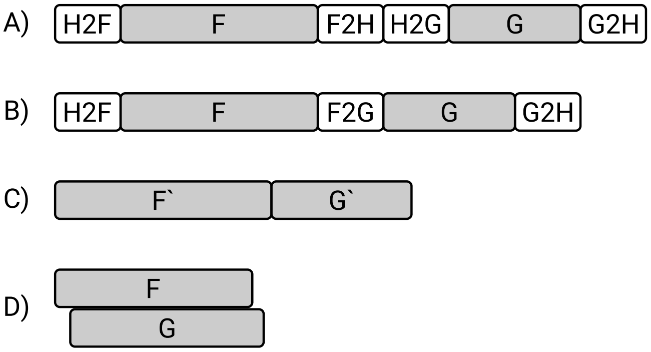


Figure 57: Tighter integration of FPGA/GPU applications. A) Current application flow. B) Application with direct accelerator communication. C) Application with CXL cacheing. D) Pipelined CXL application.

The next level of integration would be to allow the FPGA to send data directly to the GPU instead of first sending the data to the host. This replaces the F2H and H2G with a direct FGPA to GPU transfer (F2G) as shown in Figure 57b.

With the addition of CXL, the kernels are able to directly access the unified shared memory with local caches of information local to the devices. Now the execution is FPGA kernel computation F′ and the GPU kernel execution G′ as shown in Figure 57c. The kernels are denoted with ′ to indicate that the run time will increase since the data will need to be loaded into the cache. We define the disaggregation penalty as the extra time spent moving data between kernels. Thus, the disaggregation penalty can be approximated as:

$$\text{Disaggregation Penalty} = \frac{T_{total}}{T_{GPU} + T_{FPGA}} \tag{13}$$

where T_{total} is the application's total runtime, and T_{GPU} and T_{FPGA} are the total execution times on the GPU and FPGA, respectively. In order to generate these numbers, we run our application 100 times, isolate timings for the three variables, average each of them. When doing this, the disaggregation penalty becomes 1.28×, as shown in Table 13.

Finally the ideal integration would be to run the application in a data pipeline as shown in Figure 57d. This way the FPGA execution and GPU execution can be overlapped to further reduce the execution time. The speedup of the pipelined application over the current application time becomes the benefit of tighter FPGA-GPU integration. This value is 1.48× as calculated in Section 10.2 and shown Table 13.

10.1 CXL ENABLED PROGRAMMING MODEL

CXL has great potential to enhance the way heterogeneous programs are written. As discussed above and shown in Figure 57, the most performant implementation of a heterogeneous application is one which is pipelined across the multiple accelerators.

With traditional programming methods, memory buffers need to be allocated in host memory, then host-to-device and device-to-host memory transfer functions are called to move the data between the host's memory buffer and the accelerator's memory. When using multiple heterogeneous accelerators, the memory has to be moved from the previous acceleration device, to the host memory, then to the next device. This requires the programmer to explicitly manage the host and device memories and to issue appropriate host-to-device and device-to-host transfer calls. Multiple details make writing programs like this difficult because 1) the size of the data being operated on could change throughout the pipeline, 2) the different accelerators could use different programming frameworks which will are difficult to integrate together into a single tightly-integrated application, and 3) coordinating the execution of the many accelerators in a tight dataflow pattern is difficult since the host has to explicitly manage the execution of each accelerated kernel.

In addition to the CXL performance characteristics mentioned in this report, CXL's unified cache coherent memory could be leveraged to make the job of writing code targeting heterogeneous accelerators easier. Now instead of having to explicitly manage the memory buffers and memory transfers between accelerators, the devices can access the memory locations directly and the information will be automatically cached locally. Instead of explicitly managing the execution of the kernels from the host code, the host code can setup the data pipeline; then the accelerated kernels could signal with each other when they are done with a range of data and are ready for the next stage in the pipeline to start. With CXL.io the host or other devices could query the state of the accelerator and update the data rangers for the accelerator kernel to run on.

Using the ORNLapp as an example heterogeneous program, we explore how the pipelined programming flow shown in Figure 57d can be written using CXL. With the pipelined approach, the data flows from the host, to the FPGA for decompression, then to the GPU for numerical integration, and finally back to the host to store the result. The host code will read the compressed data from disk and store it into memory in chunks. The chunk size used will be a parametrizable parameter to control the granularity of the data passed in the pipeline. The FPGA and GPU kernels will be configured to run via CXL.io/PCIe device registers. These registers will control the execution of the kernel and specify where to read data from and where to store the results to, along with the data sizes. After the host stores a chunk of data into the unified address space, it will signal the FPGA kernel to start processing the chunk of data. Then the FPGA kernel can signal the start of execution on the GPU kernel once it finishes decompression. When the GPU finishes the numerical integration, it will signal to the host that the execution is done. After this, the host can write the result back out to disk. This application flow with unified memory greatly simplifies the programming model required to develop this performant heterogeneous application.

10.2 CXL APPLICATION MODEL RESULTS

The simple application model discussed in 4.8.4 is use to look at the expected performance of an pipelined application with the data broken up into various chunk sizes as shown in Figure 58. For this graph, the total application data is fixed at 1.6 GB and the chunk size is varied from a single CXL cacheline of 64 bytes to 20 KB blocks. The application data size of 1.6 GB was chosen to match the application datasize used by ORNLapp. The

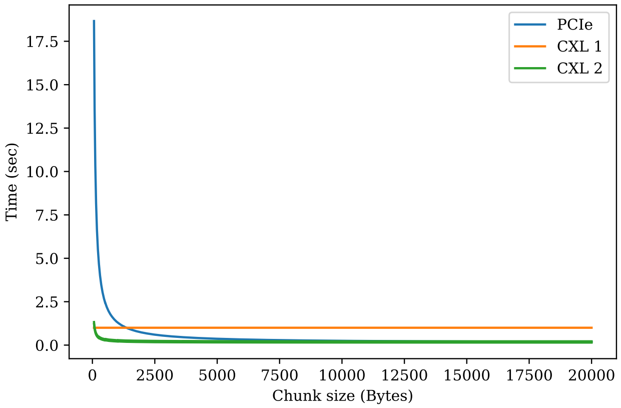


Figure 58: CXL Model: Transfer time for the entire application when transferring data in various chunk sizes. The total amount of data transferred is 1.6 GiB.

performance crossover points occur at the same location as the previous graph, with CXL model 1 outperforming PCIe for transfers less than 1.3 KiB and CXL model 2 outperforming PCIe for transfers less than 12.7 KiB. This graph highlights the potential of CXL to greatly accelerate pipelined applications with small chunk sizes. It also highlights that the minimal communication time still occurs with large chunk sizes using PCIe.

Real applications rarely have perfectly balanced pipelines, usually the pipelines are limited by the component of the pipeline which takes the most time. In the case of ORNLapp, the FPGA kernel time takes much longer than the FPGA data transfer with the kernel running for 76.2 ms and the data transfer taking 1.87 ms for reads and 1.37 ms for writes on average for each chunk size of 8 MiB for writes and 16 Mib for reads. The FPGA kernel which took the longest ran 99 times to decompress the data resulting of a total kernel time of 7.62 s. Figure 59 shows the data transfer time measured experimentally from running ORNLapp as well as the expected transfer times based on the models. The Vitis run guidance reported that the bandwidth utilization was around 60%, this correlates with the difference in runtime seen between the measured run time and the expected run time from the PCIe model. Even the slowest prediction of CXL transfer time is a seventh of the kernel run time. This means that a fully pipelined implementation of ORNLapp will be limited to the execution time of the kernel. Since the FPGA kernel also takes longer than the GPU kernel, with the total GPU kernel taking 0.27 s compared to 7.62 s the the FPGA kernel, the FPGA kernel would be the slowest part of the application pipeline. Thus the expected performance of ORNLapp if it was structured as a pipeline application with CXL could be calculated as $t_{app} = d \cdot t_{kernel} + (n-1) \cdot t_{kernel} = 7.163$ seconds. The value of $n = 99$, and it is the number of times the kernel is enqueued. The value of t_{kernel} is the slowest task in the FPGA's pipeline. The value of $d = 3$ is the depth of the application pipeline. Averaged over 100 application runs, $t_{kernel} = 0.0709$ seconds. Both are found using output data from the Xilinx tools. This is a speedup of 1.48× over the current application runtime, as shown in Table 13.

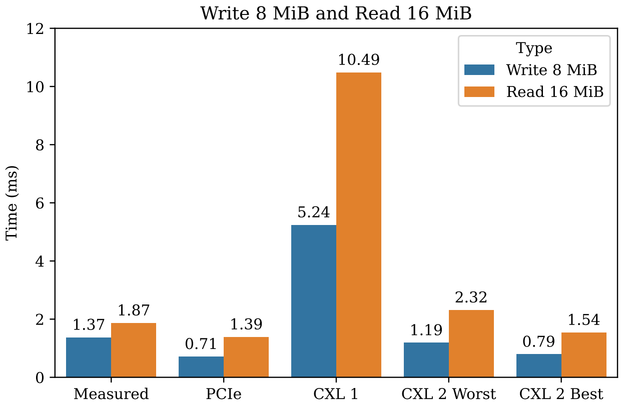


Figure 59: Comparison of actual data transfer time for ORNLapp and the times predicted by the model. CXL 2 Worst is the CXL model 2 with a 60% penalty factor. CXL 2 Best is the CXL model 2 with a 90% penalty factor.

In general, since the performance of a pipelined application is limited by the stage of the pipeline which takes the longest, the time taken for data transfers is hidden as long as the kernel execution time is greater than the data transfer time. This means that either CXL or PCIe could be used for the transfers as long as the transfer time is less than the kernel execution time. Since CXL performs well at small chunk sizes the amount of work performed at each stage of the pipeline can be reduced without the communication performance being hurt.

Not all applications can be pipelined as nicely as ORNLapp. Some have a small amount amount of work in an inner loop which could benefit from being executed on a different accelerator from the surrounding code and the dependences do not allow the loop to be unrolled into a pipelined. For this small amount of work, which we will call task t, to benefit from running on a different accelerator from the surrounding code, the task's speedup has to be greater than the penalty for moving the task's data between accelerators. Mathematically $t - t' > 2d$ where t is the time the task takes to run on the current device, t' is the time the task takes to run on a different device, and d is the time to move the data required to perform the task. The factor of 2 represents two instances of data movement: one from the current device to the target device, and then one instance in the opposite direction. If we say that the task to be accelerated is small and fits within one cache line, i.e. 64 bytes then the task time reduction required for the task to benefit from offloading the tasks is 0.08 µs. With the existing PCIe method of using accelerators, small task offloading is infeasible. To switch between accelerators the data has to be copied between devices via the host. The API overhead of moving the via the host code and the relatively large latency for small PCIe transactions prevent small tasks from benefiting from offloading to a faster accelerator. CXL could potentially change this and allow smaller portions of work to benefit from offloading to separate accelerators.

11. SUMMARY AND CONCLUSIONS

Table 13 summarizes the results for each of the integration-related topics. For consistency, we report performance relative to our NVIDIA V100 baseline. Any relative performance for integration or unified memory greater than 1.0 indicates a benefit. Conversely, any number greater than 1.0 for disaggregation is indicative of a slowdown. Not all technologies were evaluated on some applications ("N/E"). We summarize our observations, insights, and provide guidance on a technology by technology basis below.

Application	CPU-GPU Integration	CPU:GPU Ratio	Disaggregation Penalty	GPU-GPU Integration	Unified Memory	FPGA-GPU Integration	Comments
AMReX/PeleC	1.9×	2:1	5.79×	1.02×	N/E	N/E	With high implicit data movement, PeleC favors CPU-GPU integration over disaggregation
AMReX/MFIX	1.2×	1:1	2.99×	1.23×	N/E	N/E	Many small CUDA kernel calls can be a challenge for disaggregation
AMReX/CASTRO	1.0×	8:1	1.0×	1.00×	N/E	N/E	A substantial portion of parallelism is placed on the host, explaining why CASTRO favors a high CPU-GPU ratio
AMReX/WarpX	1.3×	4:1	3.79×	1.09×	N/E	N/E	Many small CUDA kernel calls can be a challenge for disaggregation
BGW: Epsilon	1.9×	3.5:1	5.7×	1.04×	N/E	N/E	The non-integer CPU:GPU ratio resulted from optimizing an analytical model. Architects should round up and/or improve relative CPU performance. GPU-GPU integration estimate assumes 2× bandwidth among GPUs within a node and 2× more GPUs per node.
BGW: Sigma	1.9×	2:1	5.8×	1.2×	N/E	N/E	GPU-GPU integration estimate assumes 2× bandwidth among GPUs within a node.
GTC-P	1.2×	3:1	2.5×	1.17×	N/E	N/E	GTCP dominated by 4 CUDA-optimized kernels
SpTRSV	1.6×	2:1	2.5×	16.6×	N/E	N/E	CPU-GPU Integration: (1) 1.6×: using eighteen GPUs over the best baseline using six GPUs. (2) 5×: using eighteen GPUs over single GPU.
SW4Lite	1.03×	8:1	1.4×	N/E	N/E	N/E	Experiments run on a Summit Node. Disaggregation experiments simulated up to 16x latency instead of 10x.
MiniVite	1.4×	1:1	5.3×	N/E	N/E	N/E	Experiments run on a Summit Node. Disaggregation experiments simulated up to 16x latency instead of 10x.
Laghos	4×	1:1	12.5×	N/E	N/E	N/E	Experiments run on a Power9 with 4 V100 GPUs (refer to Section 2.4.) Disaggregation experiments simulated up to 16x latency instead of 10x.
QuickSilver	1.03×	1:1	1.41×	N/E	0.3–0.7×	N/E	DDR-only performance vs. HBM-only performance. Experiments run on a Summit Node. Dis-aggregation experiments simulated up to 16x latency instead of 10x.
HACC	1.00×	N/E	1.10×	1.21×	N/E	N/E	Simulated by AgoChart. HACC's kernel execution time dominates its runtime.
LAMMPS	1.21×	N/E	2.39×	1.24×	N/E	N/E	Simulated by AgoChart. In LAMMPS, the number of data transfer and kernel invocations are about the same.
OpenMC	2.98×	N/E	1.45×	2.62×	N/E	N/E	Simulated by AgoChart. OpenMC invokes short-duration kernel frequently
ORNLapp	N/E	N/E	1.28×	N/E	0.6–0.9×	1.48×	Experiments run on ExCL and CXL modeled numerically.

Table 13: Summary of Integration on Performance. Note, N/E is Not Evaluated.

11.1 CPU-GPU INTEGRATION

When examining CPU-GPU integration technologies that offer a 10× increase in CPU-GPU bandwidth over PCIe and 10× reduction in CUDA launch latency, we see that half of our applications see at least a 1.3× increase in performance with a third exceeding 1.9×. For some applications like AMReX's PeleC, the benefit comes from increasing CPU-GPU bandwidth, while for other applications like AMReX's WarpX, BerkeleyGW, MiniVite, and Laghos, the benefit comes from lower average kernel launch overheads. Applications like CASTRO, SW4Lite, QuickSilver, and HACC are dominated by long-running kernels or operate almost exclusively on GPU-resident data (little CPU-GPU data movement) and are thus unlikely to see a benefit from tighter CPU-GPU integration technology.

Although we did model the impact of tighter integration on one-sided GPU-initiated communication (NVSHMEM), we made no assumptions on the benefits to MPI overheads from tighter CPU-GPU integration. If tighter integration can also improve MPI overhead, then applications sensitive to small message size performance will likely see even larger benefits. Conversely, we also made no assumption on the negative impact on CPU or GPU kernel performance due to power throttling in an integrated solution. Our estimated speedups from tighter CPU-GPU integration are likely overestimates for applications that attempt to overlap CPU and GPU computation.

Finally, all applications used in this study are already known to run well using GPUs. We did not attempt to quantify the benefit today's GPU-unfriendly applications would see on architectures that offer tighter CPU-GPU integration and thus cannot comment on the potential increase in overall NERSC/ALCF/OLCF workload throughput.

11.2 CPU:GPU RATIO

Of the 12 applications for which the ideal CPU:GPU ratio was calculated, nearly 60% require only one or two Skylake-equivalent cores per V100-equivalent GPU. This is surprisingly low given a GPU-heavy machine like Cori-GPU has five cores per V100 GPU while Perlmutter has 16 per A100. Nevertheless, such a low core count is very encouraging for integrated designs as the power, area, and pinout (quarter of CPU DDR channels) overheads of integrating two cores is very low and further facilitated with the emergence of chiplet based designs.

At the other end of the spectrum were codes like the AMReX-based CASTRO and SW4Lite both of which ran well with 8 CPU cores per GPU. When coupled with their paltry benefits from CPU-GPU integration, one might be motivated to conclude that an integrated solution is neither beneficial nor feasible. However, one should remember that for a code like CASTRO, a substantial fraction of the run time is from code running on the CPU. Such a condition amortizes the benefits of CPU-GPU integration and exacerbates the requisite CPU:GPU ratio. Rather than concluding an architecture implication, such codes might be prime targets for further optimization focused on offloading more computation to the GPU.

11.3 COMPUTE UNIT DISAGGREGATION

Although disaggregation looks like an attractive solution for performance and cost ("don't pay for what you don't use"), across our 16 applications, we see that virtually all are moderately or very sensitive to reductions in CPU-GPU bandwidth or increases in kernel launch times with a median slowdown of 2.5× and 40% of the applications exceeding a 5× slowdown from disaggregation. In fact, only 2 applications (CASTRO and HACC) saw less than a 10% impact. The former spends much of its time on the CPU while the latter has long-running kernels. In either case, the CPU-GPU interconnect is rarely used, and thus such applications are relatively insensitive to changes in CPU-GPU interconnect performance.

Any disaggregation technology must preserve the current bandwidths and launch times relative to V100 GPU performance. That is, disaggregation technologies that provide 1/3× the bandwidth to a GPU 3× faster than V100 actually replicate the same 10× negative impacts we modeled. If GPU performance improves, disaggregation bandwidth and kernel launch overheads (including long tail effects) must similarly improve. An alternate interpretation is that any disaggregation technology cannot embrace fine-grained kernel offloading, but must offload control flow and communication to the target GPU as well.

Regarding the disaggregation of FPGA and GPU, there is a 1.28× penalty that is incurred from transferring data between accelerators via CPU host memory as an intermediary step. This necessitates tighter coupling, in part, through new communication protocols like CXL in order to fully realize the compute and communication potential of truly heterogeneous systems.

11.4 GPU-GPU INTEGRATION

Integrating up to 16(12) GPUs on a node instead of 8(6) produced a median speedup of 1.2× — an artifact of the network not always being heavily exercised. In fact, only two applications exceeded 1.25× — SpTRSV and OpenMC. In the former, application performance is dominated by NVSHMEM messaging rates which see a performance cliff at the node boundary. Executing a 16-GPU solve on a single 16-GPU node replaces slow inter-node communication with fast P2P NVLINK communication on a high radix network. In the latter, OpenMC requests short-duration kernels frequently, which creates traffic congestion in loosely integrated GPUs. Thus the efficiency of OpenMC drops with higher CPU counts in loosely integrated configurations. On the other hand, tightly integrated GPUs allow the host to combine multiple command requests, which mitigates such congestion.

The performance difference between maintaining constant network injection bandwidth per-node and constant network injection bandwidth per-GPU was not evaluated.

11.5 UNIFIED MEMORY

Unified physical memory was only explored for QuickSilver where the performance of DDR-only and HBM-only solutions were compared using parameters extracted from a Summit node's V100 GPUs and POWER9 processors. We observe, depending on QuickSilver kernel, DDR-only solutions can deliver roughly 30% to 70% the performance of an HBM-only solution while concurrently providing far more memory capacity and less cost.

Unified cache-coherent memory using CXL was only explored for ORNLapp where the performance of CXL throughput was compared to PCIe. We expect CXL to deliver roughly 60% to 90% of PCIe bandwidth.

11.6 FPGA-GPU INTEGRATION

FPGA-GPU Integration explored technologies (CXL in particular) that allow for FPGAs and GPUs to directly communicate without involving the host processor/memory. Such approaches allow for a pipelined execution model. CXL provides simplified programming, cache-coherent memory, and attaining high performance. For the ORNLapp proxy app, we observe this technology could improve performance by nearly 50%.

ACKNOWLEDGMENTS

This material is based upon work supported by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-05CH11231. This research used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. This research used resources of Argonne National Laboratory, which is supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

REFERENCES

[1] Mauro Del Ben et al. "Accelerating Large-Scale Excited-State GW Calculations on Leadership HPC Systems". In: (2020).

[2] University of Bristol High Performance Computing Group. *Programming Your GPU with OpenMP*. https://github.com/UoB-HPC/openmp-tutorial. 2020.

[3] J Camier. *Laghos summary for CTS2 benchmark*. 2019. URL: https://www.osti.gov/servlets/purl/1544948.

[4] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee. 2009, pp. 44–54.

[5] Compute Express Link Consortium. *Compute Express Link Specification*. Revision 2.0. Oct. 2020.

[6] Compute Express Link Consortium. *CXL 1.1 Technical Training Videos*. Aug. 16, 2021. URL: https://www.computeexpresslink.org/cxl-regulated-videos.

[7] Jack Deslippe et al. "BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures". In: *Computer Physics Communications* 183.6 (2012), pp. 1269–1289. ISSN: 0010-4655. DOI: https://doi.org/10.1016/j.cpc.2011.12.006. URL: https://www.sciencedirect.com/science/article/pii/S0010465511003912.

[8] Nan Ding et al. "A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver". In: *Proceedings of the SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21)*. SIAM.

[9] Sayan Gosh et al. *miniVite*. 2021. URL: https://github.com/Exa-Graph/miniVite.

[10] Taylor Groves et al. *Performance Trade-offs in GPU Communication: A Study of Host and Device-initiated Approaches*. IEEE, 2020.

[11] S C Jardin et al. "The M3D-C1 approach to simulating 3D 2-fluid magnetohydrodynamics in magnetic fusion experiments". In: *J. Phys. Conf. Ser.* 125.1 (2008), p. 012044. URL: http://stacks.iop.org/1742-6596/125/i=1/a=012044.

[12] George Karypis and Vipin Kumar. "A fast and high quality multilevel scheme for partitioning irregular graphs". In: *SIAM J. Sci. Comput.* 20.1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997. eprint: https://doi.org/10.1137/S1064827595287997. URL: https://doi.org/10.1137/S1064827595287997.

[13] Ang Li et al. "Evaluating modern GPU interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect". In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2019), pp. 94–110.

[14] Weifeng Liu et al. "Fast synchronization-free algorithms for parallel sparse triangular solves with multiple right-hand sides". In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4244.

[15] Daniel Lustig and Margaret Martonosi. "Reducing GPU offload latency via fine-grained CPU-GPU synchronization". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2013, pp. 354–365.

[16] Richard C. Murphy et al. "The Structural Simulation Toolkit: A Tool for Bridging the Ar chitectural/Microarchitectural Evaluation Gap". In: 2004.

[17] Maxim Naumov. "Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU". In: *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).

[18] Rolf Neugebauer et al. "Understanding PCIe performance for end host networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 327–341. DOI: https://doi.org/10.1145/3230543.3230560.

[19] *NVIDIA Nsight Compute Profiling Tool*. https://docs.nvidia.com/nsight-compute/NsightCompute/index.html.

[20] *NVIDIA NVSHMEM Documentation*. URL: https://docs.nvidia.com/hpc-sdk/nvshmem/index.html.

[21] *NVIDIA Profiler nvprof*. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[22] Anders Peterson et al. *SW4Lite*. 2019. URL: https://github.com/geodynamics/sw4lite.

[23] David Richards et al. *Quicksilver*. 2019. URL: https://github.com/LLNL/Quicksilver.

[24] Debendra Das Sharma. *Compute Express Link*. White paper. CXL Consortium, Mar. 2019.

[25] Debendra Das Sharma and Siamak Tavallaei. *Compute Express Link 2.0 White Paper*. Tech. rep. CXL Consortium, Nov. 2020.

[26] David Skinner. "Performance monitoring of parallel scientific applications". In: (). DOI: 10.2172/881368. URL: https://www.osti.gov/biblio/881368.

[27] Nvidia Tesla. "V100 GPU architecture". In: *Online verfügbar unter http://images. nvidia. com/content/volta-architecture/pdf/volta-architecture-whitepaper. pdf, zuletzt geprüft am* 21 (2018).

[28] Oreste Villa et al. "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 372–383. ISBN: 9781450369381. DOI: 10.1145/3352460.3358307. URL: https://doi.org/10.1145/3352460.3358307.

[29] Lu Wang et al. "MDM: The GPU Memory Divergence Model". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, pp. 1009–1021. DOI: 10.1109/MICRO50266.2020.00085.

[30] Xilinx. *Vitis Accelerated Libraries*. https://github.com/Xilinx/Vitis_Libraries. 2021.

[31] Xilinx. *Xilinx Runtime Library (XRT)*. 2021. URL: https://www.xilinx.com/products/design-tools/vitis/xrt.html.

[32] Charlene Yang, Thorsten Kurth, and Samuel Williams. "Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system". In: *Concurrency and Computation: Practice and Experience* 32.20 (2020), e5547.

[33] Stephen Young et al. *FENATE*. 2021. URL: https://github.com/pnnl/FENATE.