Examining the Viability of Row-Scale Disaggregation for Production Applications

Curtis Shorts, Ryan E. Grant
Electrical and Computer Engineering Department, Queen's University
11 Union Street, Kingston, Ontario, Canada
Email: {curtis.shorts,ryan.grant}@queensu.ca

***Abstract*—Row-scale Composable Disaggregated Infrastructure (CDI) is a heterogeneous high performance computing (HPC) architecture that relocates the GPUs to a single chassis which CPU nodes can then request compute resources from. This is a distinctly different architecture from rack-scaled CDI as the GPUs are accessed over a network rather than existing in the same PCIe domain as the CPUs. Row-scale CDI expands the benefits and flexibility of rack-scaled CDI, while introducing new challenges. For example, with row-scale CDI, one must account for the effects of "slack", a latency in the CPU-to-GPU communication times due to network delays. This work seeks to assess potential challenges with row-scale CDI to determine which factors are most important to consider when deploying a CDI system. Our strong scaling application analyses reveal that there are two types of HPC workloads that may benefit from row-scale CDI; those that are CPU dominant and periodically call on the GPU to do highly parallel tasks and those that are GPU dominant and primarily rely on the CPU to coordinate work. We perform comparisons between the kernel and data transfer characteristics of each application to a slack proxy application which allowed for the development of a mathematical model to predict the performance penalty different applications can face as a result of slack. To illustrate this we profile two applications using our proposed method and find that they pessimistically would see a less than a 1% performance penalty above the effects of crossing the network in an environment which induced 100 μs of slack, or a distance of 20 km at the speed of light in a fibre optic network cable. This demonstrates that both row-scale and cluster-scale CDI are viable technologies from an application performance perspective.**

I. Introduction

Recent high demand for heterogeneous clusters (those with both CPUs and GPUs) has brought to light the pain points caused by traditional node-based high-performance computing (HPC) architectures. Traditional node architectures are constrained by physical requirements, there are only so many devices that can be attached to a physical node. For example, the number of PCIe expansion slots available to a CPU are limited, making 4 or 8 accelerators the maximum number that can be installed in a node. What if the user doesn't want or need all of the resources in a node? Compute node ratios of CPU to GPU make it impossible to perfectly match users' compute ratio needs [1], [2] at all times. This leads to complications when scheduling resources due to the 2-dimensional nature of heterogeneity [3]. The power draw of consumption of the system is inefficient as the "trapped" idle devices can't be turned off or scheduled for other jobs [1], [2]. Composable Disaggregated Infrastructure (CDI) is an emerging solution to these issues which separates the accelerators (GPUs) from the traditional node design to a pool of resources which can be composed to meet user needs. CDI allows the exact compute needs of the application to be met on-demand and enables GPUs to be to be powered down when not in use [1], [2]. This can lead to increased system efficiency for job throughput and time to solution and could also reduce the power consumption of HPC and allows more science to be performed per dollar.

Current vendor CDI solutions on the market include those from Liqid [4] and GigaIO [5] which primarily offer rack-scale disaggregated solutions (a GPU chassis only services a single rack) where the GPUs are still within the same PCIe domain. This method creates issues for scalability as rack-scale is the limit of PCIe extension technology. This limitation is admissible in smaller runtime environments such as commodity cloud services, but the scaling of HPC applications such as LLMs and large scale simulations demands a row-scale or cluster-scale (a chassis services multiple racks in one or more rows) CDI solution [1]. Row-scale disaggregation solutions like Cerio's CDI product or GigaIO's FabreX network can extend to row-scale distances. Both introduce longer paths between host CPUs and accelerators, with potential ramifications thereof.

Row-scaled CDI (henceforth CDI) is not without its drawbacks however as moving the GPUs off-node introduces a latency in the communication times between the CPUs and the GPUs as the messages now need to travel a further distance [6]. This specific type of latency as a result of CDI we define as "slack". The degree to which a specific application is impacted by slack and how much absolute slack it can tolerate, is a key question in what applications may be best suited for deployment in a CDI system. The trade-off between the performance benefits of configuring the ratio of CPUs-to-GPUs to match an application's needs and the performance hit from slack is a key metric analysing the viability of scalable CDI to row-scale and cluster-scale.

The key contributions of this paper include:

- A demonstration of the impacts of "slack" on GPU performance tested on a real system
- An analysis of the key metrics for CDI performance profiling
- A method for how the CDI metrics of an arbitrary application can be extracted without specialty hardware or any knowledge of the source code
- CDI profiles of two production HPC application's, one each from the scientific and AI/ML domains
- A discussion of the implications of the results on CDI system design and use

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00201

II. Background and Related Work

CDI is an approach to architecting computing systems where the equipment in a single compute node can be dynamically composed. Composing components means connecting them in arbitrary ways to meet the needs of a user at a given time. Instead of allocating resources as nodes in a resource manager like slurm, users instead can describe the exact resources they need and they can be composed for them in exactly the required quantities.

PCIe (Peripheral Component Interconnect Express) is an expansion bus technology. As such it has high speed and multiple lanes to connect devices to a motherboard. This means PCIe is designed for local connections of components, not long-range inter-system connections like a computer network would provide. PCIe has challenges for expansion to long-range communication, it has timeouts for transmissions, although these are long enough to potentially be avoided in today's ultra high-speed network. Another challenge is with bus enumeration as PCIe was designed to have a limited number of expansion components attached at any given time, one can run out of enumeration space on the bus. CDI providers must address these issues and have several techniques for bus enumeration either by using the entire Bus ID, device and memory space for PCIe or by abstracting them into separate PCIe domains through bus and address translation to avoid large PCIe enumerations.

Compute accelerators like GPUs are popular additions to traditional CPU node architectures as they provide methods for highly parallel data processing. GPUs are connected to hosts via PCIe connections and some vendors provide alternative intra-GPU connectivity like Nvidia's NVLink. GPUs function best with large amounts of work queued up at their scheduler, allowing the GPU to be constantly engaged in work and hide latencies associated with work setup through constant work scheduling and retirement. Latency hiding is accomplished best with plenty of work for the GPU being available, which requires host to GPU communication be fast enough to keep the GPU fed with compute (kernels).

A. Related Work

In the area of disaggregated GPUs, effort have primarily been in the field of AI for cloud computing [2], [6]. One study [6] looked in-depth at the CPU-GPU communication patterns of various AI applications and what the implications would be for their rack-scale CDI solution. The fundamental difference between cloud and HPC is the outlook on scalability with the latter requiring the full system to work efficiently as a single unit to solve the world's most important issues, such as COVID-19 therapeutics [7]. As a result, HPC could take advantages of row-scale disaggregation for the scalability of its workloads and the ability to support applications that fall both in and out of AI's domain [1].

Row-scale disaggregation is a new concept with one work [1] which addresses the topic directly. This work states the importance of row-scale CDI and how it can benefit the HPC community. Another work [8] looks at whether the statistical likelihood that intra-rack disaggregation is sufficient. The findings were that going beyond rack-scale would not be necessary, but the statistics were based on the daily use of a specific production system with a wide spread of use-cases. The authors admit in their discussion section that the contributions of the work is best attributed to their process which others can use as a framework for their own analyses.

Modeling has been done on how the disaggregation of scientific workloads can effect system utilization in HPC [9]. This modeling included LAMMPS, one of the applications in this paper, but achieved its results through mathematical extrapolation rather than experimental results. This was useful in the context of their paper which yielded a comparison of disaggregated performance in relation to other emerging architectures. This work instead seeks to mimic the delays seen in a CDI environment to analyse the secondary performance impacts that result from starving the GPU of work. The result is a better understanding of CDI's potential limitations as a technology.

GPU API remoting solutions such as rCUDA [10] can be used to run GPUs from hosts which are not in the same PCIe domain. The issue with using a remoting solution for the analysis of slack effects in a system is that they don't allow for a granular level of control over the network delays experienced. This is ultimately the result variation in the number of and time required for hops in a network being system dependent along with uncontrollable variation in network noise due to other users. rCUDA specifically is also no longer maintained under a publicly available license.

III. Experimentation Methodology

The goal of experimentation in this work is to expose the implications of CPU-to-GPU ratio scaling and the tolerance an application has for additional latency in CPU-GPU communication times added due to disaggregation (slack). The additional latency slack adds is due to the delays associated with traversing the network, as illustrated in Figure 1. The CPU-to-GPU ratio used by an application is important to understand as it provides insight into how the available compute composition is reflected in the application's performance and how traditional node architectures can under-utilize resources (both problems CDI aims to solve). Slack introduces a potential performance penalty through uncovering the latency the GPU attempts to hide behind its scheduling queue. This has the potential to counteract the benefits of a composable CPU-to-GPU ratio which is why it is important to analyse under what conditions the latency is uncovered and to what extent this can impact an application's performance.

The goal of these experiments is predictive in two aspects. The first is that row-scaled CDI hardware does not yet exist so this is predictive of the validity for CDI's value proposition. The second is that the methods used can be repeated by those who wish to implement CDI into their systems in order to predict how their specific workloads will behave in a CDI environment. To be useful for others to profile their workloads for CDI, the slack insertion model used must be implemented in software on a traditional node architecture as that is what most users will have access to. This therefore rules out using hardware methods to insert slack.

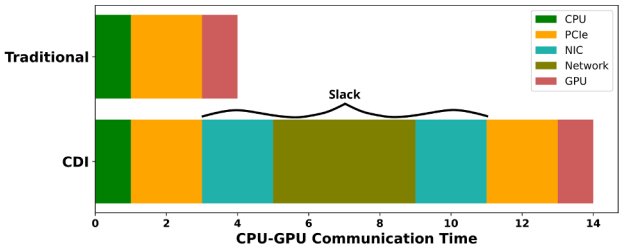


Fig. 1: An illustration of traditional and CDI CPU-to-GPU communication times. Slack is the time added by passing through the NICs and traversing network, as demonstrated by the annotation.

A. Hardware

The Digital Research Alliance of Canada's (DRAC's) Narval cluster was used which has heterogeneous nodes consisting of two AMD EPIC Milan 7413 CPUs and four NVIDIA A100SXM4 GPUs (40 GiB version) [11], [12]. The CPUs have 24 cores each for a total of 48 cores per node (12 cores per GPU). Entire nodes were reserved for experiments, even if only a portion of them would be utilized, to eliminate noise from other users [13]. All experiments were also averaged across 5 runs to account for other random effects in a system. The relevant NVIDIA ecosystem and terminologies are used in this paper as appropriate as an artifact of NVIDIA GPUs being used, but the concepts described are in no way limited to NVIDIA's GPU environment.

The point of interest for these experiments is the relation between the CPU and GPU. Therefore, a single node is used as it allows for a greater degree of control over the slack insertion. Only the ratio of the two resource types is considered, not the absolute quantity of either, which means that a single GPU can be used for developing the method. This is a result of synchronizations on the CPU side being agnostic of the GPU and vise-versa. This method is valid under the assumption that added latencies due to network channel congestion is a non-issue. The experiments that follow are designed in a way such that the worst case scenario can be considered as the total latency is the focal point.

B. Experiments

To profile the CPU-to-GPU ratio scaling of an application's performance, an application's natural affinity for each type of resource needs to be analysed on a case-by-case basis. With a single GPU and fixed problem size, we can study this by varying the number of CPU cores used. Since the problem size is fixed, this provides a basic unit of CPU-to-GPU resources can inform weak scaling for large scale production applications as the best basic CPU-to-GPU ratio.

In order to test an application's slack tolerance, we need a way of inserting slack that is scalable and easily transferable to arbitrary applications. We can achieve this by inserting an artificial delay on each CUDA API call which requires host to device (H2D) or device to host (D2H) communications. Inserting these delays manually in the source code is a prohibitively laborious and error prone process as it would need to be repeated every time a new application were to be profiled. An alternate solution is to use LD_PRELOAD which is an environment variable that allows a shared object to be loaded by the dynamic linker, ld.so, before any other shared libraries. This can be used to insert an artificial delay which mimics slack before calling a target function. Unfortunately, this approach is also not viable as it does not work for applications that rely on statically linked libraries to optimize runtime performance.

Our generalizable approach in this work is to develop a proxy application (proxy) to emulate how applications use CUDA. This allows slack to be inserted in a controlled environment where all variables can be considered. Applications can then be profiled for their slack tolerance based on where their GPU utilization characteristics fall into a spectrum of responses seen by the proxy. This requires the proxy to take inputs to control for targeted variable of interest which will be covered next. Traces of application GPU utilization characteristics are also required. NVIDIA's NSight Systems (NSys), a part of NVIDIA'S NSight suite of developer tools, [14] was used to this end. NSys was used as it is the vendor recommended tool for profiling the GPU given the variables under observation.

In order to develop a proxy for this purpose, both the CPU-GPU programming paradigm and how applications can work within it need to be explored. The typical paradigm for GPU applications is to perform compute on the CPU, transfer data to the GPU, performing compute on the GPU, transfer data to the CPU, and repeat the process. Within this paradigm, applications have the freedom to do any combination of these operations in series or parallel and to vary the amount of compute or data transfers being done. The GPU-side operations can also be done asynchronously of the CPU, but synchronous is used to capture the pessimistic case. Understanding this behavior means examining the size of a CUDA kernel, how frequently the kernels are launched, how many kernels are launched in parallel, and what the temporal offset is between parallel launches of kernels. Varying these factors will capture the range of GPU use cases seen in applications.

C. Slack Proxy Application Design

The proxy application runs a simple matrix multiplication kernel (AxB=C) using square matrices of floats. Varying the size of the matrices allows for a way to control the kernel runtimes and data transfer sizes. The main compute loop consists of copying matrix A and B to the GPU, computing C, then copying C back to the CPU. The loop is run serially for N iterations with OpenMP threads being used to implement an easy to control parallel aspect. This is done to provide complete control over the degree of the parallel component. The slack is inserted after every CUDA API call with sleep operations.

The proxy first randomly generates matrices A and B, then does a preliminary timing of the kernel to get a baseline for how long it runs. This baseline is then used to calculate work for approximately 30 seconds of raw compute on the GPU. Iteration count bounds were set to a floor of 5 and a ceiling of 1000. These parameters were placed to account for smaller kernels having larger proportional variations in runtime. The proxy uses

CPU-side control for sleeps and GPU-side control for timing. It was experimentally determined that the timing differences between the CPU and GPU side were statistically insignificant.

An additional consideration for analysing the effects of slack is the expected delay that is inserted directly by the network. This is a known effect that is experienced even in inter-node communications such as MPI collectives. The true goal of analysing application tolerance for slack is to see if there are additional delays in compute due to the GPU being starved for work. To isolate for this property, the total amount of slack inserted in the proxy needs to be removed when analysing the degradation in runtime. This is achieved by applying Equation 1 to the runtime of the main compute loop with slack and taking the delta between it and a baseline run where no slack is inserted. The number of CUDA calls per iteration of the main compute loop is 5 for the proxy as that is how many CUDA APIs are delayed (3 matrix data transfers, matrix multiplication kernel, host-device synchronization).

$$Time_{NoSlack} = Time - (num_{CUDAcalls}) * (Slack_{call}) \quad (1)$$

D. Production Applications

In this study there are general categories of applications: CPU only, GPU/CPU, and GPU dominant. The case of CPU only applications is important for CDI as trapping of GPU resources would traditionally occur with these jobs. However, no slack exists in CPU jobs as there is no accelerator. For CPU/GPU, LAMMPS was selected as the example as it is a CPU-heavy GPU enabled heterogeneous application. CosmoFlow is chosen as the GPU-dominant application. Both are production applications which are run on HPC systems and they provide comparisons for the typical case of traditional scientific applications and AI applications in an HPC environment.

1) LAMMPS: The Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [15] is a molecular dynamics simulator that simulates the bonds and forces between atoms and molecules in a crystal or container. The Lennard-Jones (LJ) benchmark was used for analysis which models short-range force interactions between identical atoms in a liquid state. LJ's simulation size is defined in 32,000 atom repeatable cubic units. The dimensions of "box size" used in later sections refers to the number of cubic units used in each Cartesian plane with a box size of 20 containing the aforementioned 32,000 units (ex. a box size of 60 is a 3x3x3 grid of 32,000 atom cubes for a total of 288,000 atoms). LJ was run for 5,000 simulation time steps in all analyses. LAMMPS was compiled with the GPU package [16]–[21] and with CUDA and OpenMPI enabled.

2) CosmoFlow: CosmoFlow [22] is a machine learning application that describes the physical model of the universe and is maintained as part of the MLPerf HPC Benchmark Suite [23]. CosmoFlow uses the TensorFlow AI/ML framework [24] to process large 3D cosmology datasets with convolutional neural networks [22]. CosmoFlow utilizes Horovod [25] for efficient communication between GPUs with MPI semantics. CosmoFlow's "mini" dataset was used, containing 1024 training and 1024 validation items, and was run for 5 epochs in all analyses. As will be seen in Section IV-B, smaller data transfers result in an application having a lower tolerance for slack which is why the mini dataset was chosen: to examine pessimistic performance case for CDI.

IV. Experimental Results

A. CPU-to-GPU Ratio

LAMMPS's affinity for CPU usage is analysed first to see to what extent a heterogeneous application's performance can rely on the CPU resources provided at a constant problem size. Figure 2 displays the results for scaling MPI processes from 1 to 24 without OpenMP threads. The runtimes are normalized in order to allow a comparison between workload sizes and to analyse the changes as a percentage. The baseline runtimes used as the normalization reference point can be seen in Table I. A box size of 20 was selected as the lower bound as it is the developers' recommended default box size for LJ. A box size of 120 was selected as the upper bound as this is the largest a single CPU core is capable of supply the GPU with data for (note the GPU's memory is not saturated at this box size).

TABLE I: Data for the different LAMMPS box sizes running with 1 process and 1 thread.

Box Size	Total Atoms	Runtime [s]
20	32k	5.473
60	288k	66.523
80	2,048k	160.703
100	4,000k	312.185
120	6,912k	541.452

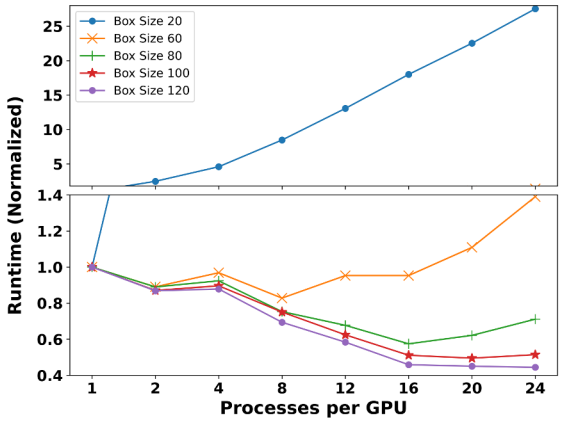


Fig. 2: LAMMPS strong scaling effects for a single GPU with an increasing number of CPUs. Note the line break corresponds to a change in scale and that the values are normalized on the y-axis.

From the results in Figure 2, a box size of 20 demonstrates the case where the problem is not large enough to see the benefit of using multiple processes as the overhead for communication is greater than the speedup of parallel compute. A box size of 60 is when this no longer becomes the case with 8 processes seeing a decrease in runtime of 17.2%. A box size of 120 benefits from 24 processes at a 55.6% decrease in runtime, although the results are diminishing after 16 processes.

The next step was to implement OpenMP threads in order to further push the upper bound of CPU utilization. 1 to 6 threads per process were tested for box sizes of 60 and up at a static 8 processes. Hyper-threading was not used so 6 threads resulted in all the cores being used. The results show that larger problem sizes saw performance benefits from using more CPU cores. The extreme for this was the box size of 120 seeing a 52.3% decrease in runtime at 6 threads versus 1; an aggregate benefit of a 76.4% decrease in runtime over the single core case.

The general trend seen from LAMMPS is that a larger problem size yields a greater need for CPU resources, but this leaves the question of what the upper bound is. An additional test was run at a box size of 200 as this saturated the GPU's memory (the way LAMMPS should be run to optimize GPU utilization in a production system). LAMMPS saw a performance increase of 24.3% with 48 CPU cores over the 24 cores case at 2 threads per process. Due to the limitations of traditional node architectures, LAMMPS would either need to run with 48 cores on 1 GPU which wastes GPU resources, or 12 cores per GPU which decreases LAMMPS performance for the CPU computation portion of the application. A CDI architecture would allow LAMMPS to request entire CPU nodes which can then each request a single GPU from a chassis thus maximizing the utilization of both resource types.

Studying the extent to which a GPU dominant application (CosmoFlow) utilizes its CPUs saw absolutely no benefits from increasing the number of processes or threads. CosmoFlow requires 2 cores to run, discovered by limiting the resources available to it. The reasons for this is hidden in the intricacies of how Python, TensorFlow and Horovod work as the reason is not directly apparent in CosmoFlow's source code and over 1000 different CPU threads were found to be in use in the application's NSys traces.

Looking again at the limitations of traditional node architectures, ComsoFlow running on four GPUs would only use up to 8 CPU cores. This wastes 40 cores which could be directed to CPU-heavy compute tasks. A CDI architecture would therefore allow for a 48 core CPU node to request up to 24 GPUs from a chassis.

B. Matrix Multiplication Slack

To start the analysis of how slack effects production applications, baseline characteristics for how and why heterogeneous applications are effected by slack needs to be extracted from the proxy. A 2D matrix size of 2^{15} squared was selected as the upper bound for proxy testing due to larger matrices not fitting on the GPU (recall from Section III-C that 3 matrices of this size are required). 1 μs of slack was chosen as the lower bound as it reflects the half-round-trip time of HPC networks with similar signalling rates as the target disaggregated technology [26], [27]. The lower bound for matrix size and upper bound for slack were determined experimentally from there. Slack insertion was run at 1 μs and the matrix sizes were decreased in multiples of 2^2 until the effects of slack were seen, thus resulting in a lower bound matrix size of 2^9. To find the upper bound for testing slack insertion, sweeps were performed at each matrix size starting at 1 μs and scaling up by orders of magnitude until slack effects above 1% were observed. The final value settled on was 10 ms at which point a matrix size of 2^{13} saw it's first increase in runtime at 10%. No slack insertion value up to 1 s was found to effect a matrix size of 2^{15}.

Table II shows the specification for the matrix sizes, the time taken for a single kernel to execute, the iteration (N) count, and the baseline runtime for the main compute loop to execute. Figure 3a) shows the slack sweep results at varying matrix sizes with a single OpenMP thread. Equation 1 is applied to the runtimes which are then normalized to the zero slack insertion case to analyse the GPU starvation effects outlined in Section III-C. Figure 3b) and c) use 2 and 8 OpenMP threads respectively to display the effects of parallelization on an application's tolerance to slack. The same data was also collected for 4 threads, but the plot is excluded for space constraints. A matrix size of 2^{15} does not fit on the GPU with 4 or 8 OpenMP threads due to each receiving their own copy of the matrices ($(3 * 4\ GiB) * 4\ threads > 40\ GiB$) and is therefore excluded from the respective figures. Experiments were also done with offsetting the time between each thread's launch and increasing the spacing between iterations of the main compute loop, but they showed no correlation to the slack performance penalty.

The key trends demonstrated in Figure 3 are that longer running kernels are more resilient to the effects of slack insertion and that the number of kernels being given to the GPU in parallel is proportional to an increase in an application's slack tolerance. Another apparent characteristic is that the "drop-off" point where slack degrades an application's performance by a large degree becomes more sharp as the slack increases.

Now that an understanding has been developed for the impacts of slack on the proxy, real applications need to be analysed to draw conclusions on how they will be impacted. This requires profiling of the applications' kernel and data movement characteristics to understand how they behave. Once this is complete, the applications can then be cross-analysed against the proxy's slack response characteristics.

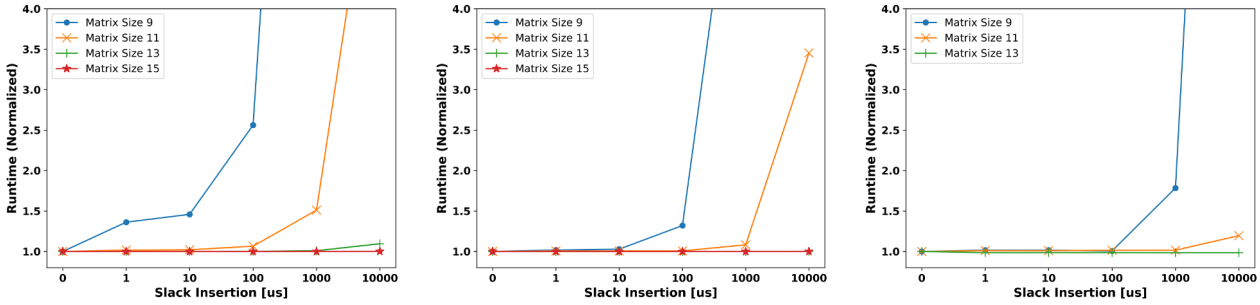
C. Application Compute Data

To extract their characteristics for comparison to the proxy, runs of LAMMPS and CosmoFlow were profiled using NSys with LAMMPS using 8 processes and 1 OpenMP thread at a box size of 120 and CosmoFlow using a batch size of 4. Traces of the proxy running at varying matrix sizes and parallel thread counts were also captured to have a sanity check to compare the application traces against. In this configuration, LAMMPS ran for 173 s and CosmoFlow for 705 s. The key metrics of interest, as seen by the proxy results in Section IV-B, were the matrix size and degree of kernel parallelization. The impact of the matrix size can be broken into its components as the kernel and data transfer portions of the main compute loop. Kernels are best characterized by their runtime whereas data transfers by the amount of data being moved.

When profiling the parallel portion of heterogeneous applications, the metric of interest is the rate at which the CPU is able to push instructions (kernels) to the GPU as the kernels

TABLE II: Data collected for the proxy running different matrix sizes. Matrix sizes and kernel runtimes are for a single matrix and single kernel. Compute loop runtime is for N executions of the main compute loop.

2D Matrix Dimensions	Matrix Size [MiB]	Kernel Runtime [us]	Serial Kernels (N)	Compute Loop Runtime [s]
2^9	1	120	1000	0.382
2^{11}	16	4,714	1000	8.121
2^{13}	256	362,710	83	35.112
2^{15}	4096	23,150,700	5	120.679



a) 1 parallel thread. b) 2 parallel threads. c) 8 parallel threads.

Fig. 3: Matrix multiplication results for varying OpenMP thread counts.

are queued for execution on the GPU side. The launching of kernels is done by LAMMPS with parallel processes, 8 of them in the traces collected (which is still a small number as seen in Section IV-A). The processes start the kernels then wait for them to complete. CosmoFlow queues its processes in a different manner, using a small number of cores. CosmoFlow submits a large number of varying sized kernels in quick succession which then form an instruction sequence. The CPU then performs other tasks in the background and waits for the sequence to complete. The launching of the kernels takes approximately 1/7th the duration of the sequence (these happen in parallel) so a pessimistic parallelization of 4 will be used for the comparison in Section IV-D.

The distribution of each application's kernel durations is shown in Figure 4. The distributions are shown for each kernel as well as the total for all kernels to gain more insight into the frequency and duration of each of the application's operations. Only the top five kernels are shown for CosmoFlow as it executes dozens of different ones. These five account for 49.9% of the total runtime with the aggregate effects of the remainder being captured in "Total" as seen in Figure 4.

Figure 5 shows the distributions of data transfer sizes for each of the applications. The memory behaviour of both applications is consistent with the expectations set by the kernel distributions and trends seen in the NSys traces.

D. Application to Proxy Comparison

Now that the proxy and applications are characterized for their kernel durations and data transfer sizes, a model can be developed for how to predict the performance penalty an application could see with a CDI network. This model is captured in Equation 2 where the kernel and memory slack penalties (SP) are drawn from comparing the proxy results to those of the applications and where the % runtimes are the proportion of the applications' runtime where kernel and memory operations are being performed respectively.

$$SP_{Total} = \%\ Runtime_{Kernel} * SP_{Kernel} + \%Runtime_{Memory} * SP_{Memory} \quad (2)$$

To compare the data transfer sizes in Figure 5 to the matrix size values in Table II, a binning of the application transfer sizes can be done as presented in Table III. This is also done for the kernel runtimes in Figure 4, but was excluded due to space constraints. Equation 3 uses this binned data as the element counts for each matrix size and the slack penalty for each matrix size from Figure 3(a-c). As discussed in Section IV-D, LAMMPS uses 8 parallel process and CosmoFlow has the equivalent queuing effect of 4 processes so the slack penalty values are drawn from Figures 3(c) and the data that wasn't plotted for 4 parallel threads respectively.

$$SP_{Kernel|Memory} = \sum_{MatrixSizes} SP_{Matrix\_Size} * Elements_{Matrix\_Size} \ /\ Total\ Elements \quad (3)$$

TABLE III: A binning of the data transfer sizes for LAMMPS and CosmoFlow in MiB.

	< 1	< 16	< 256	< 4096	> 4096	Mean
LAMMPS	2264	42016	40008	0	0	16.85
CosmoFlow	8186	668	335	640	0	34.49

The final results of entering all of these values into Equations 2 and 3 are shown in Table IV. There are two total slack penalty values due to the binning of applications' data falling in a range between two matrix sizes. This results in a lower and upper

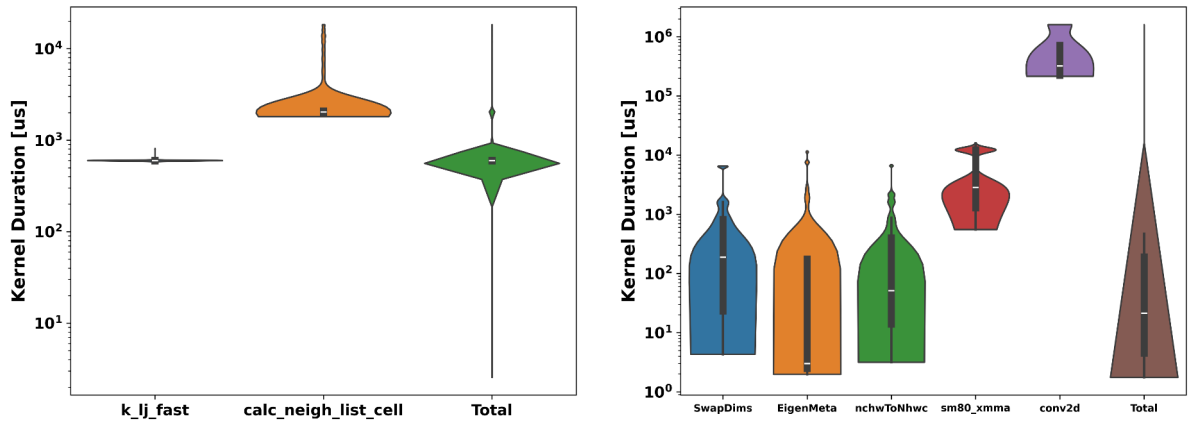


Fig. 4: Violin plots for the kernel durations of LAMMPS (left) and CosmoFlow (right).

TABLE IV: The upper and lower total slack penalty results for LAMMPS and CosmoFlow at varying slack values.

	Slack [us]	1	10	100	1000	10000
LAMMPS Total Slack Penalty	Lower Value	1.000	1.008	1.009	1.025	1.417
	Upper Value	1.011	1.014	1.009	1.539	11.486
CosmoFlow Total Slack Penalty	Lower Value	1.002	1.002	1.004	2.871	24.501
	Upper Value	1.004	1.004	1.005	3.384	30.772

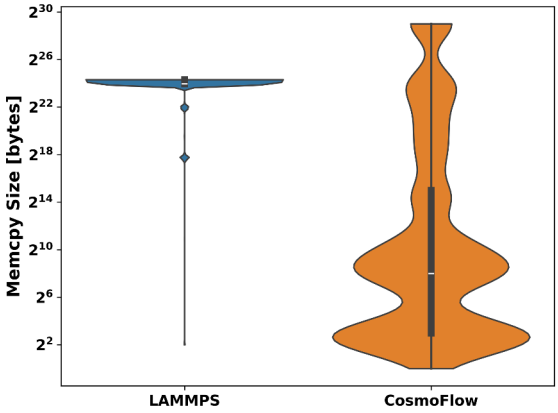


Fig. 5: Violin plots for memcpy sizes.

value for the slack penalty for each application depending on if the matrix size equivalents are rounded up or down respectively.

This methodology for comparing the applications to the proxy was validated using the proxy NSys traces and seeing how closely the traces predicted their own performance. The final results for the single threaded runs were that the lower value was within 0.005 the actual and the upper value resulted in a severely pessimistic estimate of the slack penalty. The more threads that were added the less pessimistic the upper value became as the exponential slack response became less of a factor. Preliminary tests were also done with the LD_PRELOAD method for artificial slack insertion described in Section III-B from which the results generally agreed with those seen from the methods we used, but as mentioned previously complete confidence in coverage of API calls is difficult.

The main takeaway from Table IV is that both applications can pessimistically see a less than a 1% decrease in performance from 100 μs of slack. This is a very high bar for a penalty given that the half-round-trip time of a modern HPC network is on the order of 1 μs. Looking at the time from another perspective, light can travel up to 20 km in a fibre optic cable in this amount of time which suggests that newly emerging CDI solutions could be datacenter scale instead of just rack or row scale if the other challenges with enumerating large numbers of devices can be overcome.

V. Discussion

In this section we answer some outstanding questions regarding CDI and the implications of the results.

Why does the slack performance cost follow an exponential curve? Amdahl's Law. The cost to do parallel compute increases with CDI so the fall-off point for the performance benefits decreases the longer the serial barrier (slack) becomes. The extreme example of this is would be if the slack were so large that the GPU became completely idle while waiting on the CPU. In this case, the GPU can't reap from the benefits of pipelining its operations. This is the same reason why Equation 1 was used to remove the direct effects of the delay; the cost of communicating over the network is admissible, but the point at which the GPU is starved for work is when the benefits of parallelization are lost.

I thought CDI was bad for performance? The benefit of CDI is in the ability to configure the system to the exact compute needs of the application. There can be a performance benefit associated with this when GPUs are located physically close to one another, but incurs overheads in CPU to GPU communication.

In the case of LAMMPS, there's a need for a large number of CPUs in proportion to GPUs for which the results of have been measured. In the case of CosmoFlow, the want is for as many GPUs as possible to accelerate the task. Where fitting 16 GPUs in a single node is not possible with a traditional node architecture, CDI can allow for this to be the case in a single GPU chassis, which can greatly increase the performance of CPU asynchronous operations such as GPU-to-GPU collective operations.

Why use CDI for GPUs? GPUs are the most expensive aspect of the nodes in a heterogeneous system and require peripheral components to be added such as CPUs. CDI provides the architect with a choice with how to allocate their resources based on their needs. Add another M racks of heterogeneous nodes to the system, each with N GPUs and their respective peripherals, or disperse M chassis throughout the system with 2N GPUs each to provide additional GPU resources on-demand. As PCIe chassis can be inexpensive per PCIe slot compared to traditional node architecture, this provides a compelling alternative system architecture.

Why do LAMMPS and CosmoFlow perform differently? The difference in the application's performance fundamentally come down to the problem each is solving. LAMMPS is performing scientific operations which are mostly serial compute on the CPU side, but can benefit from offloading largely parallel tasks in the pipeline to the GPU. CosmoFlow on the other hand is propagating data through convolutional neural networks which is a GPU oriented task.

Both applications can ultimately benefit from CDI through the optimized scheduling of resources. Taking an example where 40 GPUs and 20 CPUs, each with 24 cores, are available in a system and both LAMMPS and CosmoFlow want 20 GPUs. A traditional node architecture would schedule both LAMMPS and CosmoFlow with 120 CPUs per node. A CDI architecture would instead be able to allocate CosmoFlow with 4 CPUs to control its 20 GPUs in a chassis. This enables CosmoFlow's GPU-to-GPU collectives to perform faster as the GPUs are more closely coupled. This would leave LAMMPS with the remaining 16 CPUs which makes the CPU-to-GPU ratio 5:4, much better for LAMMPS compute needs than the previous 1:2.

VI. Conclusions

Row-scale CDI is a developing technology that allows for CPUs and GPUs to be configured in a ratio which best optimizes application performance and compute resource utilization. This paper presented a method for testing the validity of CDI as a technology by exploring the advantages and drawbacks it offers for the performance of production applications. The method provided is executable in software on any traditional node architecture without administrative access which provides prospective adopters of CDI with the ability to test the value proposition of it for the specifics of their use case. The findings were that the benefits of CDI outweigh the performance penalty introduced by slack when the CPU is able to provide the GPU with a sufficient amount of work. This can be achieved through the use of proportionally long running kernels or by sending a large number of short running kernels to the GPU. Both production applications studied were able to achieve this up to 100 μs of slack.

These results demonstrate that a GPU does not rely as heavily on tight coupling to its CPUs as was once believed. This opens the question from beyond rack-scale to cluster-scaled CDI as the speed of light limitation for 100 μs results in a distance of 20 km before accounting for other network effects. Future work seeks to validate the results of this method through testing on CDI hardware once the technology is available. Testing will also be required for the characteristics of CDI which may bottleneck application performance outside of the compute characteristics explored in this work.

VII. Acknowledgement

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant RGPIN 05389-2016 and Compute Canada. Computations were performed on the Narval supercomputer at McGill University.

Thank you to Ali Farazdaghi, Olasupo Ajayi, and Tooraj Taraz for your insightful conversations and providing feedback on this work.

References

[1] M. Williams and R. E. Grant, "The future of ai/ml innovation is row-scale disaggregation," in OCP Global Summit, San Jose, CA, 2023.

[2] A. Guleria, J. Lakshmi, and C. Padala, "Emf: Disaggregated gpus in datacenters for efficiency, modularity and flexibility," in *2019 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2019, pp. 1–8.

[3] M. Sheikhalishahi, R. M. Wallace, L. Grandinetti, J. L. Vazquez-Poletti, and F. Guerriero, "A multi-dimensional job scheduling," *Future Generation Computer Systems*, vol. 54, p. 123–131, Jan 2016.

[4] Liqid, "Composable infrastructure software platform," 2024. [Online]. Available: https://www.liqid.com/

[5] GigaIO, "Cloud-class composable disaggregated infrastructure," Apr 2023. [Online]. Available: https://gigaio.com/project/microchip-and-gigaio-on-cloud-class-composable-infrastructure/

[6] B. He, X. Zheng, Y. Chen, W. Li, Y. Zhou, X. Long, P. Zhang, X. Lu, L. Jiang, Q. Liu, D. Cai, and X. Zhang, "Dxpu: Large-scale disaggregated gpu pools in the datacenter," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 4, p. 1–23, Dec 2023.

[7] H. Lee, A. Merzky, L. Tan, M. Titov, M. Turilli, D. Alfe, A. Bhati, A. Brace, A. Clyde, P. Coveney, H. Ma, A. Ramanathan, R. Stevens, A. Trifan, H. Van Dam, S. Wan, S. Wilkinson, and S. Jha, "Scalable hpc & ai infrastructure for covid-19 therapeutics," *Proceedings of the Platform for Advanced Scientific Computing Conference*, Jul 2021.

[8] G. Michelogiannakis, B. Klenk, B. Cook, M. Y. Teh, M. Glick, L. Dennison, K. Bergman, and J. Shalf, "A case for intra-rack resource disaggregation in hpc," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1145/3514245

[9] T. Groves, C. Daley, R. Gayatri, H. A. Nam, N. Ding, L. Oliker, N. J. Wright, and S. Williams, "A methodology for evaluating tightly-integrated and disaggregated accelerated architectures," in *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2022, pp. 71–81.

[10] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," in *2010 International Conference on High Performance Computing Simulation*, 2010, pp. 224–231.

[11] Digital Research Alliance of Canada, "Béluga," 2019, last accessed 16 September 2017. [Online]. Available: https://docs.alliancecan.ca/wiki/B%C3%A9luga

[12] ——, "National host sites," 2024, last accessed 21 January 2023. [Online]. Available: https://alliancecan.ca/en/services/advanced-research-computing/federation/national-host-sites

[13] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System noise, os clock ticks, and fine-grained parallel applications," *Proceedings of the 19th annual international conference on Supercomputing*, Jun 2005.
[14] Nvidia, "Nvidia developer tools," 2024. [Online]. Available: https://docs.nvidia.com/nsight-developer-tools/
[15] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "Lammps - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, Feb 2022.
[16] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - short range forces," *Comp. Phys. Comm.*, vol. 182, pp. 898–911, 2011.
[17] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers - particle-particle particle-mesh," *Comp. Phys. Comm.*, vol. 183, pp. 449–459, 2012.
[18] W. M. Brown and Y. Masako, "Implementing molecular dynamics on hybrid high performance computers – three-body potentials," *Comp. Phys. Comm.*, vol. 184, pp. 2785–2793, 2013.
[19] T. D. Nguyen and S. J. Plimpton, "Accelerating dissipative particle dynamics simulations for soft matter systems," *Comput. Mater. Sci.*, vol. 100, pp. 173–180, 2015.
[20] T. D. Nguyen, "Gpu-accelerated tersoff potentials for massively parallel molecular dynamics simulations," *Comp. Phys. Comm.*, vol. 212, pp. 113–122, 2017.
[21] V. Nikolskiy and V. Stegailov, "Gpu acceleration of four-site water models in lammps," *Proceeding of the International Conference on Parallel Computing (ParCo 2019), Prague, Czech Republic*, 2019.
[22] A. Mathuriya, D. Bard, P. Mendygral, L. Meadows, J. Arnemann, L. Shao, S. He, T. Karna, D. Moise, S. J. Pennycook, K. Maschoff, J. Sewall, N. Kumar, S. Ho, M. Ringenburg, Prabhat, and V. Lee, "Cosmoflow: Using deep learning to learn the universe at scale," 2018.
[23] MLPERF, "Benchmark mlperf training: Hpc: Mlcommons v2.0 results," Jan 2024. [Online]. Available: https://mlcommons.org/benchmarks/training-hpc/
[24] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, R. Jozefowicz, Y. Jia, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, M. Schuster, R. Monga, S. Moore, D. Murray, C. Olah, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow, Large-scale machine learning on heterogeneous systems," Nov. 2015.
[25] A. Sergeev and M. D. Balso, "Horovod: fast and easy distributed deep learning in TensorFlow," *arXiv preprint arXiv:1802.05799*, 2018.
[26] J. Liu, J. Wu, S. P. Kini, D. Buntinas, W. Yu, B. Chandrasekaran, R. M. Noronha, P. Wyckoff, and D. K. Panda, "Mpi over infiniband: Early experiences," 2003.
[27] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.