An illustration of extending Hedgehog to multi-node GPU architectures using GEMM

Nitish Shingde[1,2*], Timothy Blattner[3], Alexandre Bardakoff[3], Walid Keyrouz[3], Martin Berzins[1,2]

[1*]SCI Institute, University of Utah, 201 Presidents Circle, Salt Lake City, 84112, UT, USA.
[2]Kahlert School of Computing, University of Utah, 201 Presidents Circle, Salt Lake City, 84112, UT, USA.
[3]NIST, 100 Bureau Drive, Gaithersburg, 20899, MD, USA.

*Corresponding author(s). E-mail(s): nitish@sci.utah.edu;
Contributing authors: timothy.blattner@nist.gov; a.bardakoff@prometheuscomputing.com; walid.keyrouz@nist.gov; mb@sci.utah.com;

Abstract

Asynchronous task-based systems offer the possibility of making it easier to take advantage of scalable heterogeneous architectures. This paper extends the previous work, demonstrating how Hedgehog, a dataflow graph-based model developed at the National Institute of Standards and Technology, can be used to obtain high performance for numerical linear algebraic operations as a starting point for complex algorithms. While the results were promising, it was unclear how to scale them to larger matrices and compute node counts. The aim here is to show how the new, improved algorithm inspired by DPLASMA performs equally well using Hedgehog. The results are compared against the leading library DPLASMA to illustrate the performance of different asynchronous dataflow models. The work demonstrates that using general-purpose, high-level abstractions, such as Hedgehog's dataflow graphs, makes it possible to achieve similar performance to the specialized linear algebra codes such as DPLASMA.

Keywords: Hedgehog, MPI, multi-node GPU

1 Introduction

Continuing innovations in hardware pose challenges to developing portable software, particularly for new heterogeneous architectures. These challenges may be addressed by the adoption of new programming models for efficient compute node use that should represent parallel constructs and make it easier to instrument and reason about an application's performance, thereby allowing developers to gain deeper insight. Two examples of such models are the Hedgehog software [2], and the Uintah Computational Framework [6, 14]. Hedgehog specializes in a single compute node level performance based on C++ threads and NVIDIA CUDA. Uintah specializes in large-scale simulations and uses an MPI+X hybrid parallelism model. This paper is an extended form of the conference paper [1] and shows how it uses Hedgehog mixed with the general philosophy of Uintah to obtain performance on the well-studied problem of dense matrix-matrix multiplication (GEMM) in a distributed environment. The whole Section 2, first three parts of Section 3, and Subsection 4.1 are taken from the conference paper [1] while the last two Subsections from section 3 and from section 4.2 onwards are the extensions of that work.

The central idea is to use GEMM to demonstrate how Hedgehog provides a way for rapidly prototyping high-performance code based on dataflow graphs. In the conference paper [1], we demonstrated the efficacy of the Hedgehog system with a simple matrix multiplication algorithm (HHDG1). The HHDG1 version showed how well Hedgehog could give comparable results against DPLASMA and SLATE but only for moderately large enough matrices and a small number of compute nodes. For this reason, the next version (HHDG2) of the GEMM algorithm is modeled on DPLASMA using Hedgehog. The HHDG2 version presented here uses a 2D block cyclic data distribution for all the matrices, the same as DPLASMA. This paper also introduces the concepts of *Jobs* and *computation windows*, which are used to break down the workload distribution, similar to DPLASMA's computation blocks. The *Sender* and *Receiver* tasks are used in HHDG1 to establish a communication line between the local graphs on each compute node. Similarly, a new task, DataWarehouse, is introduced and used in HHDG2 and explains how it serves a different use case than the one before.

The rest of the paper is organized as follows. Section 2 discusses the various frameworks that deal with multi-GPU distributed-memory platforms. This section also talks about existing state-of-the-art techniques used for tackling GEMM operations. Section 3 presents the design principles used to implement GEMM for a multi-GPU accelerated distributed-memory platform in Hedgehog. Section 4 discusses and compares the GEMM algorithm implemented in Hedgehog for both the versions, HHDG1 and HHDG2. Section 5 compares Hedgehog's results against those of DPLASMA. Section 6 ends the paper with a conclusion and future plans.

2 Existing approaches

2.1 HPX

HPX [20] (for High Performance ParallelX) is a C++ library for exascale computation. Exascale computing is an architecture dealing with the parallelism and communication

Table 1: Key Notations

Notation	Explanantion
M, K, N	matrix size A(M, K), B(K, N), C(M, N)
T	tile size
M_T, K_T, N_T	matrix sizes expressed in tiles
$p \times q$	MPI grid dimension
$gp \times gq$	GPU grid dimension
G	GPUs per compute node
d	depth of chunck
l	look ahead parameter
W_W, W_H	window dimensions

between nodes achieving 10^{18} Tflops. HPX AMT (Asynchronous Many-Task) runtime system presents an API conforming to the C++ standard for local or remote computation, and implements an asynchronous execution model that semi-automatically parallelizes user code.

HPX uses C++ futures to transform sequential algorithms into asynchronous executions with a wait-free property; computation and communication can be overlapped with the usage of C++ 20 `co_await` operator. They have created "local control objects" (LCO) for synchronization mechanisms. One of them, the data-flow, allows the execution of a piece of code on a separate thread when the values that it depends on become available. The thread usage allows to get a minimal overhead for synchronization and context switching.

The HPX scheduler comes with a work stealing algorithm and an automatic load balancer. HPX has its own C++ implementation of the C++ 17 algorithms and C++ 20 concurrency facilities. Their work served the design for the C++ parallelism technical specification [1]

2.2 Legion

Legion [21] is a task graph library for heterogeneous nodes. The library defines "logical regions" in memory as collections of objects. When creating a task, the end-user defines explicitly the task's input data plus the attached properties. The properties include the logical regions privileges (read, write, or both), the region organization (if it is an array of structures or a structure of arrays), the partitioning and the coherence (the types of treatments that a task can do in another task's logical region). The runtime system will use these different logical regions' properties to define the scheduling between these tasks and handle tasks reordering. It can also duplicate shared read-only logical regions to improve parallelism between multiple tasks.

2.3 StarPU

StarPU [22] is a C library to execute parallel tasks over heterogeneous hardware on a single node with a task graph representation. It proposes a unified approach (a

[1] https://stellar-group.org/2016/03/hpx-and-cpp17/. Compared to other AMT systems, HPX brings a "future-proof C++ conforming API" and an exposed asynchronous programming model.

codelet) to implement a task. End-users can use additional libraries like Nvidia CUDA [23] or BLAS routines to implement the kernel inside the same codelet. The codelet will be then offloaded to the execution unit used, CPU or accelerators. In addition to the unified execution model, StarPU proposes a generic scheduling framework. It enables users to customize low-level scheduling decisions, such as work stealing or work balancing, with high-level calls or per-task performance models.

2.4 Charm++

Charm++ [24] is an object-oriented parallel message passing programming system based on C++. It follows an *asynchronous many-task* model where an application is decomposed into transferable units of work with their inputs. These units are called *chares*. They start their execution upon reception of a message. These chares are presented under the form of C++ objects with all the particularities brought by the language; they present encapsulation of data and inheritance capabilities. A chare will do its computation based on the data it embeds and transferred input data. Data safety is guaranteed by only running one chare on a piece of data. The execution system will associate each chare to a different execution unit thanks to a dynamic scheduler to maximize the performance. The overall workflow is described in a "charm interface" cross-compiled into C++ code [25]. They claim portability without changes on all MIMD (multiple instruction, multiple data) computers.

2.5 Uintah

Part of the original motivation for the extension of the Hedgehog system to multiple compute nodes is the scalability of asynchronous many-task (AMT) runtime systems and their use in helping manage the increased concurrency, deep memory hierarchies, and heterogeneity. Such runtime systems are advantageous for their ability to handle increasing compute node level parallelism through the task overdecomposition of an application while also managing low-level system details necessary for efficient resource utilization behind-the-scenes. Examples include Charm++ [12], HPX [11], Legion [5], PaRSEC [7], and Uintah [6].

While Uintah has demonstrated large scale scalability on heterogeneous architectures [14], it started as a fixed task-graph execution code and was extended to dynamic task execution [13]. Uintah's runtime system manages the asynchronous and out-of-order (where appropriate) execution of these tasks and addresses the complexities of (global) MPI and (per compute node) thread-based communication. Execution is managed by the task scheduler, which interacts with per-MPI process task queues to select and execute ready tasks (e.g., tasks with satisfied data dependencies). In extending Uintah to heterogeneous architectures, Kokkos [8], was used to meet the challenges posed by diverse heterogeneous systems. Uintah application code then is decomposed into individual tasks that are executed on either the host or device and that make use of Uintah's intermediate portability layer [10], with options to use Kokkos. The resulting tasks are then compiled into a task graph and dynamically executed by the heterogeneous runtime system in an asynchronous out-of-order manner. Scaling capabilities have been shown for two benchmarks using Uintah's MPI+Kokkos scheduler [9] and the accompanying portable abstractions [10] to execute workloads representative of typical Uintah applications. The recent results [14] show good, strong-scaling to 24,576 NVIDIA V100 GPUs and 8,192 IBM POWER9 processors and demonstrate Uintah's preparedness for the diverse heterogeneous systems accompanying Exascale computing. The key lessons from Uintah for this work are to use separate task graphs per MPI process and use the global dependency map stored in Data Warehouse to prioritize external communication while hiding its impact using overdecomposition.

2.6 DPLASMA

DPLASMA is a distributed parallel linear algebra software targeted toward multi-core architectures. The matrix multiplication algorithm uses the Parameterized Task Graph (PTG), a type of Domain Specific Language (DSL), and exposes it in a compact and problem-size independent format that is queried on-demand to discover data dependencies in a distributed fashion. It depicts algorithms using data flow principles as pure data dependencies between BLAS kernels. The resulting dataflow depiction uses PaRSEC, a state-of-the-art runtime system, to run it in a distributed environment. The algorithm uses several control dependencies like b and c (block sizes for matrix C), d (depth), and l (look-ahead) to increase the data reuses and optimize the communication flow from/to accelerators within each compute node. It uses cuBLAS's General Matrix Multiplication (GEMM) kernel for computation and MPI for nodal communication.

2.7 SLATE

Software for Linear Algebra Targeting Exascale, also known as SLATE, aims to provide newer linear algebra packages targeting modern many-node HPC clusters. It uses a newer matrix storage format where tiles are the first-class objects, thus leaving the traditional dense linear algebra software like ScaLAPACK, Elemental, and DPLASMA to use contiguous memory to represent the local matrix in each process. SLATE uses a collection of individual tiles to represent the matrices, with no correlation between the tile's position in the matrix versus in memory. SLATE uses MPI for distributed node parallelism, OpenMP for explicit thread parallelism within nodes, implicit thread parallelism within the vendor's node-level BLAS, and SIMD vector instructions for vector parallelism. SLATE relies on explicit dataflow information for communication, where it will broadcast the required tiles to the processes where it is needed. This approach yields a multicore performance of 170 TFLOP/s on 16 nodes and a peak accelerator performance of 339.2 TFLOP/s when processing double-precision matrices [4].

2.8 Hedgehog

Hedgehog [17] is a C++ header-only library without any dependencies for developing general purpose coarse-grained parallel algorithms. It targets a heterogeneous single-node compute units with one or multiple CPUs and one or multiple GPUs. Its execution model works without any added scheduler; the inner threads, attached to Hedgehog nodes, are managed only by the operating systems, and execute based on the presence of data.

The multiple inputs need to be expressed in some way with C++. The language feature that hedgehog uses is variadic templates.

The Hedgehog nodes are attached with edges representing the flow of data using queues that store unprocessed data. A node can have multiple input and output edges. Hedgehog uses the C++ variadic template feature to explicitly define and manage separate input queues. The user needs to explicitly connect the input type of one node to the output type of another node to define a dependency between the two nodes for that particular data type. If no type is specified, a dependency is created for all common types. Also, if the type specified by the user is invalid or there is no common type, then an error is raised at compile time. The nodes and edges are structured in the form of a dataflow graph. These nodes are independent persistent entities that accept and produce data. A node starts its execution as soon as input data values are available. Because a node can be linked to another node and each of them are living on different threads, they form an inherent parallel asynchronous data pipeline. This pipeline is used to get performance: it simplifies parallelizing I/O, data motion, and computation, and it maximizes system utilization by leveraging data streaming. This implementation aims to design portable performing graphs for heterogeneous nodes (e.g., featuring multiple GPUs).

Hedgehog operates with a variety of nodes. Multi-threaded tasks are responsible for doing heavy computation. These tasks form a group, and share the same input and output edges consisting of queues and synchronization contexts. State manager tasks use a localized state, which is thread-safe shareable environments, used for data synchronization. A graph is also a node, allowing graph composition and code sharing. This separation of concerns is considered as a first-class citizen as it facilitates the programmability of the library.

Diverse metaprogramming techniques secure the graph by checking its consistency and validity at compile-time. It is also possible to build a compile-time representation of the graph allowing user-defined tests execution on this representation while compiling and consequently modifying the outcome of the compilation.

Bardakoff et al. have demonstrated the performance of this approach with single compute node computations in [2]. The Hedgehog LU decomposition with partial pivoting performed on par with the Linear Algebra Package (LAPACK) dgetrf routine compiled with OpenBLAS in multi-threaded mode. For the matrix-multiplication (BLAS-like GEMM routine), running specific matrix sizes, Hedgehog achieves > 95 % of theoretical peak across 4 NVIDIA V100 GPUs, outperforming cuBLASMg and cuBLAS-XT baseline libraries.

2.9 Comparing Task-Based Runtime Systems

The number of task-based codes is growing quickly and so there have been multiple examples of comparisons of these approaches with the aim of explaining their advantages and disadvantages. An early and detailed comparison of Legion, Charm++ and Uintah was made by a DOE team [29]. The OpenMP, HPX and Legion runtime Regent are compared in [26] with a suggestion that on CPU architectures HPX and OpenMP are perhaps more efficient. One issue in all task based systems is that system overhead may be an issue. Wu et al. [31] compare the overheads of Charm++ and HPX.

The study shows that there is room for improvement in both systems. There are fewer comparisons of these systems on GPUs, however Gu and Becci compare MPI, open Shmem, Charm++ and Legion on dense matrix multiplication (DGEMM) with matrices of rank 2400 or 4800, among other things. They note that "delegating memory management, synchronization handling and load balancing hurts Charm++ and Legion performance and scalability". These automated capabilities are, however, part of the attraction of these systems. There are comparisons of more specialized linear algebra approaches on both CPUs [28] and on GPUs [29], in the latter case achieving about 90% of peak. There is a discussion of the design of Hedgehog and a discussion of some alternative approaches in the thesis of Bardakoff [17]. One of the main contrasting features of Hedgehog from some other approaches is that it uses a very light and fast threading approach that has the potential to achieve high node performance. Traditional task-based runtime systems schedule worker threads given a queue of tasks. Hedgehog, on the other hand, operates using a scheduler-free dataflow approach where persistent tasks are bound to threads. These threads awaken and execute only when data is sent to them. The lightweight nature of Hedgehog was demonstrated on a single node, achieving >95 % of the theoretical peak across 4 GPUs in matrix multiplication [2]. Hedgehog maintains a static graph during compilation, execution, and profiling. Data flows through this graph dynamically based on demands, and metadata about execution per element is captured. This style of execution makes profiling much simpler as it keeps the task graph from blowing up, in contrast to Legion, where the graph size could get out of hand at times. This work is an attempt to exploit these capabilities and to see if a more general runtime system can achieve the same levels of performance on multi-node DGEMM compared to more specialized linear algebra approaches.

3 Extending Hedgehog to Multiple Compute Nodes

Hedgehog executes the dataflow graph entirely scheduler-free based on the flow of data. The order in which this execution model passes data to tasks is non-deterministic, relying entirely on the order in which the operating system context switches threads. This out-of-order design is a staple in how Hedgehog obtains performance but poses some design challenges for getting performance on distributed systems. For example, typical MPI programs expect a structured approach that embeds a specific ordering of messages between nodes. Additionally, Hedgehog nodes are designed in its model for non-overlapping usage to achieve a separation of concerns. For instance, the state manager in Hedgehog is a specialized task that manages the state between two or more tasks. We follow the same separation of concerns design and maintain Hedgehog's execution model when augmenting Hedgehog's abstractions to support multiple compute node scaling with three new specialty tasks: (1) *Sender*, (2) *Receiver* and (3) *DataWarehouse* tasks. Similarly to Uintah [6], each compute node has its own local task graph instead of having one global task graph to manage work across the compute nodes for scalability. Each of these local graphs contains these two new specialty tasks to establish a form of communication. While the intra-node communication has been automated out by Hedgehog based on [2], the inter-node communication is an active area of research for extending Hedgehog to multiple nodes. In Section 3 the *Sender*, *Receiver* and *DataWarehouse* tasks are implemented specifically for matrix multiplication and deal with point-to-point communication. This inter-node communication is established via these special tasks instead of making it part of the Hedgehog to prevent prematurely getting tied to any one particular approach, albeit at the cost of inconvenience. The design adopted here attempts to extend Hedgehog to multiple nodes with the aim of preserving the high performance of Hedgehog on nodes. This requirement requires an approach that leaves almost all of the nodal architecture intact while having the flexibility to potentially scale across multiple nodes. Other systems like Uintah [13], Legion [21], HPX [20], PaRSEC [21] have automatic inter-nodal data movement, whereas here it is more explicit since this is in the early stages of extending Hedgehog to Multiple Compute nodes. Although these tasks use MPI underneath as their communication framework, they are designed to be agnostic of such communication models.

The *DataWarehouse* task and the *Sender*, *Receiver* tasks both serve different use cases. The *DataWarehouse* task does not have the dependency map, which makes it suitable for applications that make undetermined data transfers. The drawback of this approach is that it can be slow since it is a 2 step process, first exchanging the metadata and then communicating the messages. This cost can be hidden by prefetching the tiles from matrices A and B as much as possible. The *Sender*, *Receiver* tasks are suitable where the user already knows when and where the data is needed, i.e., it has the dependency map. This tactic was beneficial in the previous version [1] since we knew where to send and reduce the partial results for matrix C.

Hedgehog also supports another special task, the execution pipeline, which duplicates a graph for multi-GPU computations within a single compute node. The task functions as a way to execute across multiple GPUs within the same process, reducing the inter-process communication that is often used when utilizing multi-GPU machines. The implementation abstracts the complexity of sending work to each GPU by automatically duplicating a graph into sub-graphs and binding each sub-graph to a GPU. The user then just needs to implement a decomposition strategy to describe how data is sent to each sub-graph. This paper builds on this concept by expanding the capabilities of Hedgehog to bind a graph per node in a cluster of nodes and defines a new *Job* abstraction that is used to define different decomposition strategies for multi-node execution, as described in the section below.

3.1 DataPacket

Serialization/deserialization of data converts complex data structures into a byte stream and vice versa. DataPacket has a buffer to help store these byte streams. We define a MatrixTile class that composes and uses the DataPacket class to store the tile's metadata and the two dimensional matrix-tile data for matrix multiplication. By making DataPacket part of the MatrixTile, we use the DataPacket's buffer to store and use the metadata and data directly. This helps circumvent the overhead of allocating a new DataPacket object and copying the serialized bytes from the tile to the DataPacket.

3.2 Sender Task

A *Sender* task processes data from within the graph and sends them to *Receiver* tasks across processes/compute nodes. The incoming data to the sender task specify the destination compute node; the sender does not implement any logic to decide where the message should go. In addition to sending the message, it also sends a context ID as metadata. In MPI, this is possible in the form of tags. The context ID helps the receiver task to deduce the type of message. In the case of matrix multiplication, the "output state" feeds the accumulated tile along with the destination compute node for the Sender task to pack the data into a DataPacket and send it across to the *Receiver* task of the receiving compute node.

3.3 Receiver Task

Similar to the *Sender* task, the Receiver task registers all possible data types involved in inter-node communication in the form of template parameters. As discussed in section 2.8, this is how Hedgehog establishes an edge or a dependency between two nodes for intra-node communication, and the same construct is used for extending the inter-node communication as well. The Receiver task is a daemon thread, which polls for any incoming messages without actually receiving the message. The Receiver task obtains the context ID from the polling (tags in MPI), deduces the appropriate data type and buffer size, and enqueues an asynchronous receive call for the incoming message. The receiver task periodically checks this queue for any completed received messages, and based on the data type, it deserializes and pushes the data out through the appropriate outgoing edge. These connections are established when adding edges in the graph between the receiver tasks and their endpoints. The data flows automatically to the correct endpoint using its data type, which is instrumental in how Hedgehog operates its edges. The Receiver task is defined in this way in order to handle the out-of-order execution and handle spurious sends based on the flow of data within other processes. There is room for improvement in this approach as the daemon becomes a thread that periodically sleeps. One potential optimization will be if a communicator uses a monitor-based implementation when sending/receiving messages, which would allow for the receiving thread to enter into a wait state until a message is incoming.

3.4 DataWarehouse Task

Having a preexisting knowledge of data dependency makes the prefetching tasks like *Sender* and *Receiver* quite efficient. However, it is unreasonable to have such expectations for every use case. As influenced by similar concepts and approach used in Uintah [6], the *DataWarehouse* task, a flexible server-client fashioned task, was implemented to fill this void. Each compute node has its own local Hedgehog graph. Figure 1 shows how the *DataWarehouse* task is added to each of those graphs to establish a communication line between different compute nodes. This task has a global map of where the data exists but does not have the dependency knowledge, i.e., when and where the application needs the data at any given point. The *DataWarehouse* task processes two types of requests: intra-node and inter-node requests. When the *DataWarehouse* task receives the intra-node requests, it checks the global map where the requested data resides. The task outputs the requested data if the data is available locally. If the data resides on another compute node, then the *DataWarehouse* task sends an inter-node request to that node to initiate the data transfer. This inter-node request is relatively small in size, and sending too many such requests can blow up the queue and slow down the whole communication process. Hence, the task is designed to accept a batch of inter-node requests to lower the frequency.

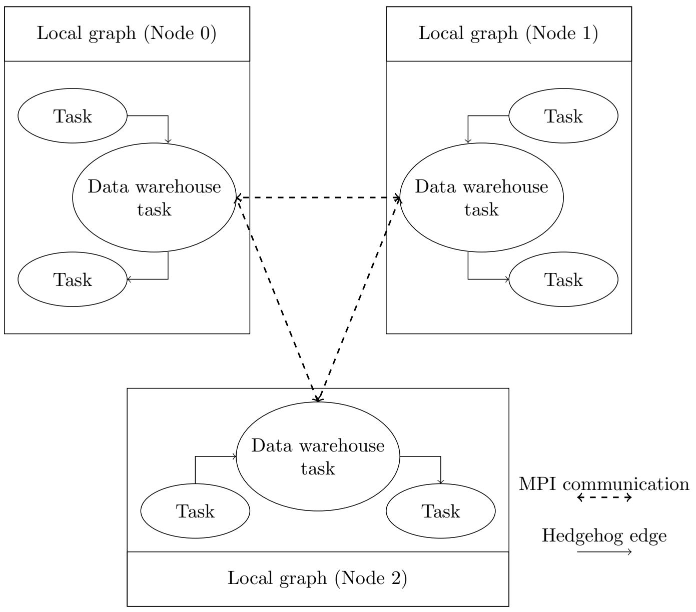

Fig. 1: Data Warehouse Task

3.5 Job abstraction for Multiple Accelerators

```
// Static round robin filtering based on the col index of Tile A
bool sendToGraph(std::shared_ptr<Tile> tileA, int gpuId) override {
    return tileA->colIdx()%gpuCount == gpuId;
}

// Static round robin filtering based on the row index of Tile B
bool sendToGraph(std::shared_ptr<Tile> tileB, int gpuId) override {
    return tileB->rowIdx()%gpuCount == gpuId;
}
```

The HHDG1 implementation statically distributed the workload via round-robin based on the K_T index. This approach [1], as seen in the pseudo-code above, tightly coupled the workload distribution with the design of the dataflow graph itself. To adopt a different workload distribution, we also need to change the dataflow graph. Hence, to make it more flexible, we introduced the abstraction of a *Job* where we can describe the workload distribution. The *Job* defines the computation domain that can be scheduled on any available device. The pseudo-code below depicts how a job is used like a filter, which can discard the incoming data irrelevant to that particular *Job*. The ability of the *Job* abstraction to dynamically schedule Jobs and filter data makes it useful for fast prototyping. In the HHDG2 implementation, the *Job* describes a set of tiles (computation window) from matrix C for work. Based on the indices of the matrix C tiles, the relevant tiles from matrices A and B are requested. The set of matrix C tiles defines the Job or the workload distribution, and it could be chosen in a round-robin, 2D block-cyclic, or some arbitrary method, and the filtering will still work accordingly.

```
// Graph filter is set dynamically based on the submitted job}
bool sendToGraph(std::shared_ptr<Tile> tileA, int gpuId) override {
    return job[gpuId].rowIndices.contains(tileA->rowIdx());
}

// Graph filter is set dynamically based on the submitted job
bool sendToGraph(std::shared_ptr<Tile> tileB, int gpuId) override {
    return job[gpuId].colIndices.contains(tileB->colIdx());
}
```

4 Matrix Multiplication using Hedgehog

Subsection 4.1 gives details about the version of the algorithm implemented from the conference paper [1]. The later subsections discuss the current version of the algorithm, the data and workload distribution used, the change in data movement complexity, and expected scalability.

4.1 Previous version: HHDG1

The algorithm implemented here is an extension of the single compute node setup implemented in section 4.3 of Bardakoff's thesis [17]. The thesis explores the algorithm's evolution from CPU only to CPU+GPU to CPU+multiple GPUs using Hedgehog. We briefly revisit the single compute node setup and then its subsequent evolution to multiple compute nodes using the abstractions mentioned in section 3. While the approach used here lays down the general approach to extend Hedgehog to multiple compute nodes, the communication model used here is hardwired to this case for matrix multiplication. While the peer-to-peer and one-sided communication requirement is more aligned with Hedgehog's design principle, it makes scaling more challenging, which needs to be addressed in future work.

The terms M, N, and K represent the dimensions of the matrices. T represents the tile size, and M_T ($\lceil \frac{M}{T} \rceil$), N_T ($\lceil \frac{N}{T} \rceil$), and K_T ($\lceil \frac{K}{T} \rceil$) represent the number of tiles along the M, N, and K dimensions of the matrices, respectively.

4.1.1 Single compute node setup

Figure 2 highlights the data and work distribution. Each matching pair of columns and rows from matrices A and B depicts a unit of work per GPU.

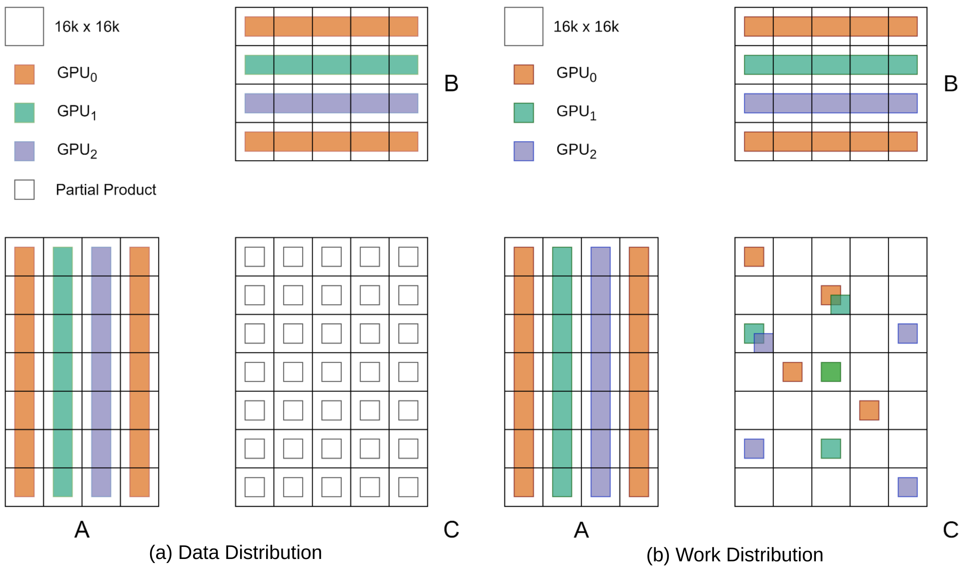

Fig. 2: Fig. (a) represents the data distribution. For each GPU, only 1 column of tiles from A and 1 row of tiles from B are considered at a time. For matrix C, each GPU gets p partial product tiles (reusable), for storing the partial GEMM computations. Fig. (b) represents the work distribution on the GPUs. It is quite similar to the data distribution, where each GPU calculate the partial result for all the elements in matrix C

The workload is offloaded to each GPU in a round-robin fashion to ensure equal distribution of work. Tiles from matrices A and B are copied to the respective GPUs, where all the tiled-GEMM kernel execution occurs. One thing to note here is that all the GPUs work independently. As we use the outer-product approach, each unit of work asynchronously outputs a partial result for the whole matrix C in the form of tiles. These tiles, called product tiles, are copied back to host memory from the GPU memory for accumulation with matrix C. The accumulation is done on the CPU. There are $M_T * N_T * K_T$ such tile accumulations, i.e., $M * N * \frac{K}{T}$ addition operations in total. It is important to note that the factor $\frac{K}{T}$ here keeps these CPU-side accumulation tasks from being the bottleneck. The GPU memory needs to be large enough to accommodate M_T tiles from a column of matrix A, N_t tiles from a row of matrix B, and 4-8 tiles for storing the product tiles. For detailed information on the Hedgehog data flow graph and its working, refer to section 4.3.1 from Alexandre's thesis [17].

In Hedgehog, the task graph is instantiated only once during its creation. When a task receives new data, the data simply waits in a queue if all the threads concerning the tasks are busy. This differs from traditionally used task graphs in systems like StarPU [15], PLASMA, and CILK [16], where the directed acyclic graph (DAG) gets unrolled as it keeps receiving data. The actual performance in this approach comes from pipelining the memory copies and kernel execution tasks using NVIDIA's streams and asynchronous API calls. The CUDA streams help synchronize the host-to-device memory copies of tiles from matrices A and B, cuBLAS GEMM kernel execution using those tiles, and device-to-host memory copy of the product tiles outputted by the kernels.

4.1.2 Multiple compute node setup

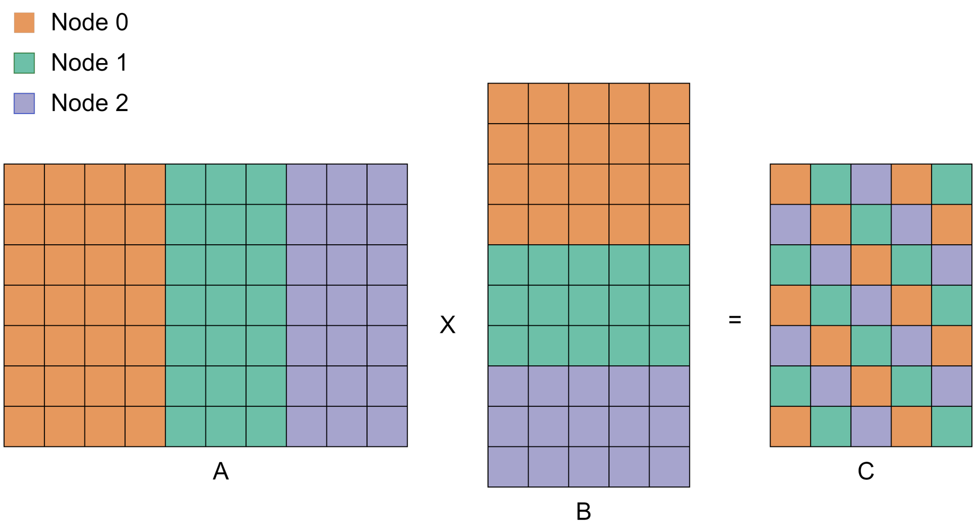

Fig. 3: Data distribution of matrices across multiple compute nodes. Matrices A and B are distributed in a 1D Block Column and 1D Block Row fashion respectively. Matrix C, as a whole, redundantly resides on all the compute nodes with the ownership marked in 2D Block cyclic fashion.

Figure 3 highlights the data distribution in a multi compute node setup. Matrices A and B are partitioned in a 1D column and row block-cyclic fashion, respectively. This nature of the data distribution allows us to treat these sub-matrices of A and B as matrices themselves and use the previous single compute node setup to independently compute partial results for every element in matrix C. In the current design, every compute node calculates a partial result for all the elements in matrix C. We need to reduce the matrix C present on each compute node to get the final result. There are two types of accumulations happening here, one within a compute node, which we will simply call accumulation, and the other is inter-node, which we will call reduction, to help distinguish between the two. The cost of reducing matrices is significant and grows as the matrix size and/or the number of compute nodes increase. The accumulation of matrix C tiles (within a compute node) happens in stages. So instead of waiting for the whole matrix C to get accumulated, we asynchronously send the accumulated tile as soon as it is ready. Figure 3 depicts the round-robin target distribution of the tiles in matrix C. This distribution of matrix C helps evenly distribute the sends and receives. Using this approach helps spread the communication cost over the execution of the hedgehog graph instead of dealing with a costly singular reduction call. To achieve this asynchronicity, we use the sender and receiver task approach, as detailed in Section 3. For the receiver task we had first-hand knowledge of the type of messages and their count from the beginning. Since only 1 type of message was involved, namely, the tiles from matrix C, we could skip the polling step and directly initiate/enqueue an asynchronous receive call.

4.2 Current version: HHDG2

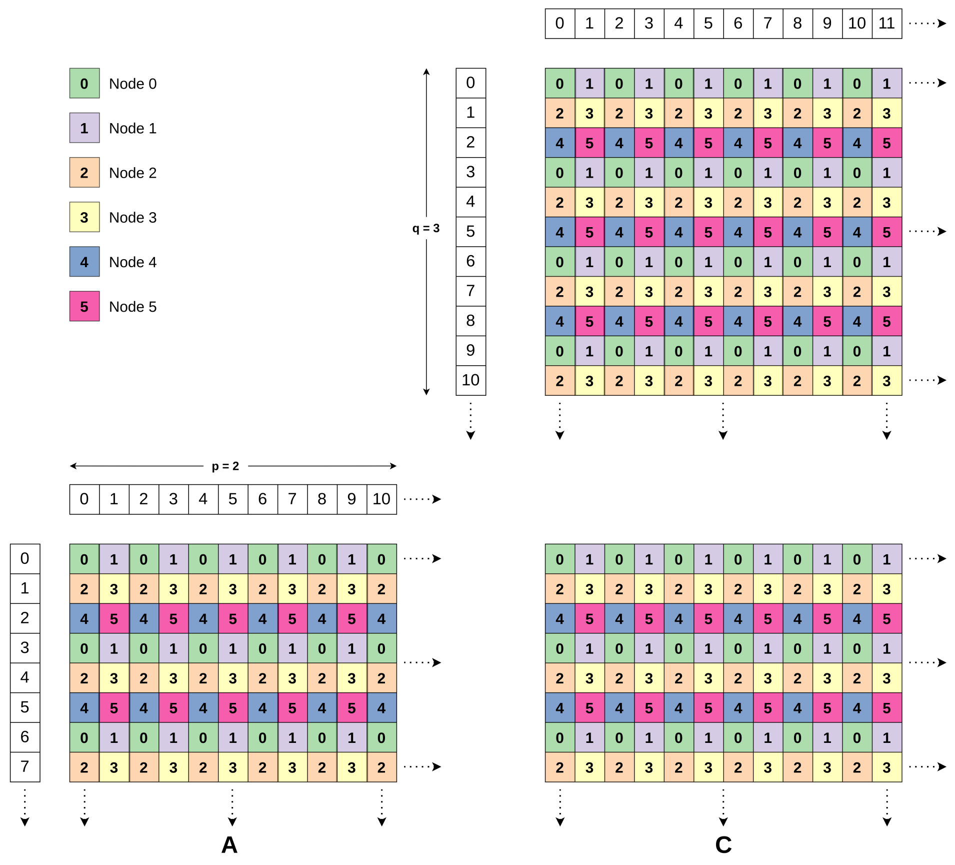

Fig. 4: Multiple Compute Node Data Distribution

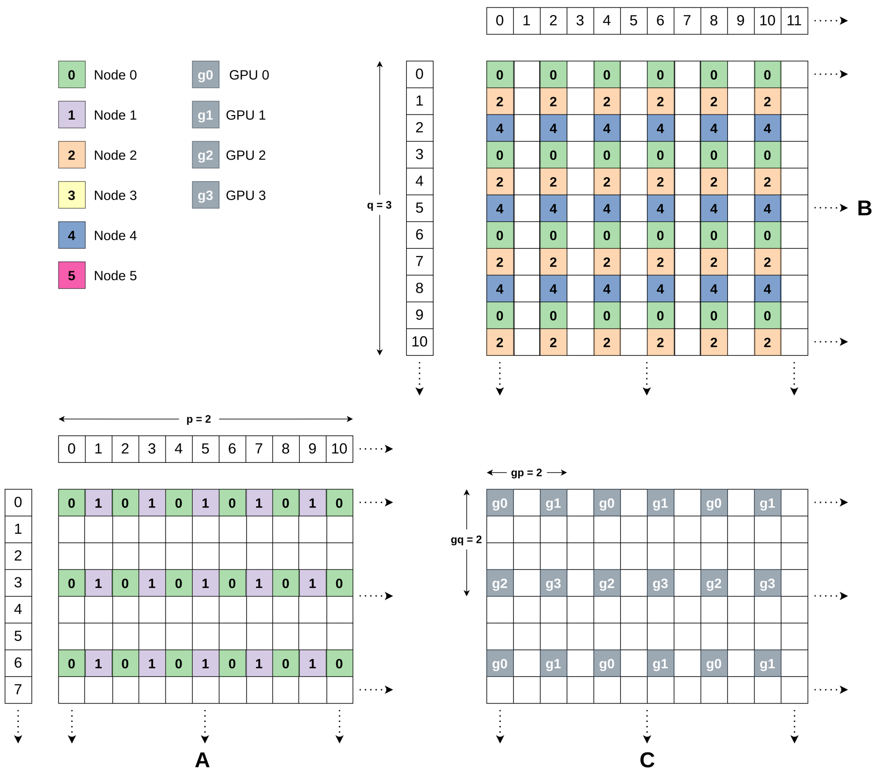

Fig. 5: GPU Workload Distribution on Node 0

The HHDG1 algorithm attempt [1] using Hedgehog to apply matrix multiplication on a distributed architecture gave comparable results to existing software such as SLATE and DPLASMA. Matrices A and B were distributed in a 1D column and row block-cyclic fashion, respectively, while matrix C was distributed in a 2D block-cyclic fashion. The data distribution of matrices A and B was such that each compute node was self-sufficient in terms of getting the input data; however, the algorithm computed partial results for the entire matrix C on each compute node and reduced these partial results to get the final solution. This inherent requirement of redundant matrix C on each compute node made it harder to scale for larger matrices and more compute nodes. Hence, a newer approach was adopted based on DPLASMA's philosophy. Instead of moving the data from matrix C across multiple nodes while keeping matrices A and B local, matrix C was kept local while moving the data from matrices A and B as required. As depicted in Figure 4, a 2D block-cyclic data distribution of all the matrices is adopted. This version also does not have the concept of product tiles since both the partial product and the accumulation are done on the GPU.

4.2.1 Work load distribution

Workload distribution is done by partitioning the matrix C, since that is where the computation takes place. The computation window is just a set of tiles which has W_W, W_H number of tiles along the width and height. Figure 5 shows how the computation is divided across GPUs on a compute node. This is partitioned further if the GPU memory capacity is not big enough. The optimal window size is computed based on three factors: the number of GPUs available on a compute node, each GPU's memory capacity, and the matrix C's size. Similar to the grid of compute nodes $p \times q$ used in data distribution, we define a grid of GPUs $gp \times gq$. For example, if there are four GPUs, there are three possible GPU grid configurations: (1, 4), (2, 2), (4, 1). The GPU grid dimensions are accepted as a user parameter. The workload is partitioned such that the number of windows is a multiple of the GPU grid dimension to ensure equal distribution of work as much as possible. An exhaustive/brute-force search is conducted to find the largest possible window dimension, such that all the tiles from matrix C ($W_H \times W_W$) within the window plus the tiles required from matrix A ($W_H \times d$) and B ($W_W \times d$) with certain depth d can fit inside the GPU memory $W_H * W_W + d \times (W_H + W_W) \leq GPU_{tile\ capacity}$. DPLASMA computes the window size such that the tiles from matrix C occupy at most 75% of the GPU memory capacity and uses the rest of the memory to store tiles from matrices A and B.

4.3 Algorithm

The computation is divided as per the distribution mentioned in the above subsection. This version uses an outer product approach for matrix multiplication. Each GPU is assigned a job that needs to do the computation on the tiles from matrix C within the assigned computation window. All the jobs within a compute node need to be synchronized, as the tiles from matrices A and B are batched and prefetched such that it keeps the MPI communication volume to a minimum. These batches are, however, prioritized based on the number of tiles that are already available locally. Local tiles are sent first to avoid idling the GPU while waiting for the other tiles to be prefetched. All the tiles from matrix C within the computation window ($W_H \times W_W$) are copied to the GPU, along with a $d \times W_H$ and $d \times W_W$ number of tiles from matrix A and B, where d represents the depth of input tiles to be brought on to the GPU. The depth parameter is a nonzero integral parameter, ranging from 1 to K_T, often depending on the GPU capacity and matrix sizes. As this version uses the outer product approach, the algorithm needs W_H row tiles from matrix A and W_W column tiles from matrix B to compute. The depth parameter also determines the window computation size due to the following constraint $W_H * W_W + d \times (W_H + W_W) \leq GPU_{tile\ capacity}$, and as the d parameter increases, the computation window size ($W_H \times W_W$) decreases, leading to too many creation of small jobs which can dampen the overall performance. The parameter d is defaulted to two, where depth one is supposed to do computation and depth two for prefetching the next set of tiles. Like DPLASMA, we use the look-ahead parameter l to decide the level of prefetching. On each compute node, $l \times d \times W_H$ and $l \times d \times W_D$ number of tiles are allocated for prefetching tiles from matrix A and B, respectively. Depending on the memory constraints, the user can choose the parameters l and d, with one and two recommended values, respectively. Hedgehog's memory manager is used in enforcing these data decomposition limits. This is a mechanism to keep the dataflow graph from oversubscribing system resources while also keeping the graph busy to overlap computation with IO. Charm++ [24] has a similar system where they use the GPU manager, which uses user provided buffers and CUDA kernels to execute and profile such tasks on the GPUs. The GEMM operation and accumulation are both then done on the GPU, unlike the previous HHDG1 version, where the GEMM was done on the GPU and the accumulation was done on the CPU. This approach adds the complexity of carefully scheduling the GPU workload to avoid race conditions but reduces the intra-node communication significantly compared to the previous one.

4.4 Data movement complexity

Data movement plays an important role in achieving performance. In this section, we will compare both, the inter-node communication volume using MPI and the intra-node communication volume between host memory and GPU memory.

4.4.1 Inter-node complexity

In the previous version, no inter-node communication occurred for matrices A and B. The only communication that takes place is for matrix C. The communication volume is equivalent to a collective reduction call, which is $M \times N \times (n-1)$, where n is the number of compute nodes.

For the current approach, matrix C is stationary, whereas tiles from matrix A and B are communicated redundantly based on the computation window. The communication volume for matrices A and B depends on how the matrix C is partitioned for computation. Similar to the MPI grid dimension, we have grid dimensions for GPU as well, gp and gq. Matrix C is partitioned as a multiple of these GPU grid dimensions a and b, where the ideal case is when a and b are both equal to 1. These a and b are chosen such that the tiles from matrix C within the window plus tiles from matrices A and B up to a certain depth d can fit entirely inside the GPU. The communication volume for matrix A is $q \times M \times K \times b$, and matrix B is $p \times K \times N \times b$, where $1 \leq a \leq \lceil \frac{M_T}{p \times gp} \rceil, 1 \leq b \leq \lceil \frac{N_T}{q \times gq} \rceil$.

It is hard to compare the communication volume for this version to the previous version, as the previous version was not dependent on the capacity of the GPUs. For easy comparison, consider the case of square matrices. The inter-node complexity of the newer version HHDG2 ranges from $2\sqrt{n} \times N^2$ to $(n+1) \times N^2$ while the previous version HHDG1 has $(n-1) \times N^2$. In the worst case, the HHDG2 version has $2 \times N^2$ more. The worst case in the HHDG2 version implies the window size is of $(1, 1)$, i.e. just one tile of matrix C, which is highly unlikely.

4.4.2 Intra-node complexity

In the previous version, only tiles from matrices A and B are transferred to the GPU. In total, across all the compute nodes, $M_T \times K_T$ tiles from matrix A, and $K_T \times N_T$

tiles from matrix B are transferred from the host memory to the GPU memory. The partial computations are stored in an uninitialized memory in GPU, called product tiles. These product tiles are computed and copied from GPU memory to host for $M_T \times N_T \times K_T$ times. Therefore, the total communication volume, in terms of tiles, is $K_T \times (M_T + N_T + M_T \times N_T)$. For square matrices, this is equal to $N_T^3 + 2 \times N^2$.

For the current version, we copy the tiles from matrix C from host to GPU and vice versa once, $M_T \times N_T \times 2$. Whereas for tiles from matrices A ($M_T \times K_T \times b$) and B ($K_T \times N_T \times b$) are copied from host to GPU across all compute nodes, where $1 \leq a \leq \lceil \frac{M_T}{p \times gp} \rceil, 1 \leq b \leq \lceil \frac{N_T}{q \times gq} \rceil$. The total intra-node complexity comes out $2 \times M_T \times N_T + K_T \times (b \times M_T + a \times N_T)$. For square matrices this is equal to $N_T^3 \times (\frac{1}{p \times gp} + \frac{1}{q \times gq}) + 2 \times N_T^2$, where $1 \leq p, 1 \leq q, 1 \leq gp, 1 \leq gq$. By comparing the coefficient of the highest power term (N_T^3), the current version ($\frac{1}{p \times gp} + \frac{1}{q \times gq}$) gets smaller compared to the older version (1) for larger p, q, gp and gq. This means that as the number of compute nodes and GPUs increases, the intra-node complexity of the HHDG2 version decreases, dropping below the HHDG1 version.

4.5 Expected Scalability

The HHDG2 implementation is quite similar to that of DPLASMA, as both of them use the same data distribution and computation strategy, with the most significant differentiating factor being DPLASMA using Parsec [7] and HHDG2 using Hedgehog [2]. Another key difference between the two is the workload distribution. As mentioned in section 4.2.1, DPLASMA calculates the compute window size such that matrix C occupies 75% of the GPU memory capacity at the most. DPLASMA does a round-robin column-wise distribution of tiles from matrix C on the GPU. The HHDG2 implementation, on the other hand, tries to find the window size which satisfies the condition $W_H * W_W + d \times (W_H + W_W) \leq GPU_{tile\ capacity}$ and W_H and W_W divides the matrix C such that it is a multiple of the grid dimension $gp \times gq$. The workload is distributed in a 2D block-cyclic fashion on the GPU. Both implementations try to synchronize the GPU workload to overlap the MPI communication and thus minimize the inter-node data movement complexity. DPLASMA has shown to scale [3] on the Summit with 432 GPUs for a matrix size of around 1 million elements in each dimension. The close similarities between DPLASMA and HHDG2 open the possibility of achieving similar scalability when such experiments can be conducted.

5 Results

The results in sections 5.1 and 5.2 talk about different environment setups the experiments were run on. All the experiments depicted are evaluated for square matrices of varying sizes made of single precision floating point numbers. Every run is measured over ten times and presented as mean and standard deviations of the execution times (seconds) and performances (TFLOP/s).

5.1 Single compute node

The single node setup consists of an Nvidia DGX Compute Node with 2 Intel Xeon E5-2698 v4 2.2 GHz 20 Core Processor with Hyper-Threading (80 Threads per Node), 512 GB of RAM, and 8x Tesla V100 GPUs. The experiment was compiled using gcc/9.5.0 and cuda/11.7 The Tesla V100 GPUs have a peak performance of 15.7 TFLOP/s per GPU and, therefore, a peak performance of 125.6 TFLOP/s for the entire DGX node. Table 2 shows performance for three single precision matrix sizes: 128K, 160K, and 192K, each achieving 78.7%, 77.1% and 84.9% of the peak performance, respectively. The DPLASMA could not run for matrix size 192K as the experiments ran out of memory.

Table 2: DGX Compute node

	M = N = K = 128K		M = N = K = 160K		M = N = K = 192K	
Algo	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s
DPLASMA	44.4 ± 0.2	101.5 ± 0.4	88.1 ± 0.5	99.9 ± 0.6	-	-
HHDG2	45.6 ± 0.8	98.9 ± 1.76	90.9 ± 1.7	96.8 ± 1.8	142.6 ± 0.9	106.6 ± 0.6

Note: DGX compute node is equipped with8 V100 GPUs. The whole compute node has a peak performance of 125.6 TFLOP/s.

5.2 Multiple compute nodes

The Kingspeak cluster [19] setup at the University of Utah consists of four nodes. Each node has two 14-core Intel Broadwell processors (E5-2680 v4 running @ 2.4 GHz), 256 GB of RAM and 2 Tesla P100-PCIe 16GB GPUs. The cluster is configured with Mellanox FDR Infiniband interconnect which has a bandwidth of 57 Gbit s^{-1}. Each GPU has a peak performance of 9.3 TFLOP/s, and hence the total node configuration has a peak performance of 74.4 TFLOP/s. The code was compiled using gcc/10.2.0, Intel MPI/2021.1.1 and cuda/11.6.2. Table 3 shows performance results for 3 single precision square matrix sizes: 128K, 192K and 256K, each achieving 89.5%, 91.4% and 91.8% of the peak performance, respectively.

Table 3: Kingspeak

	M = N = K = 128K		M = N = K = 192K		M = N = K = 256K	
Algo	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s
DPLASMA	67.9 ± 0.2	66.3 ± 0.2	226.4 ± 1.3	67.2 ± 0.4	534.1 ± 1.0	67.5 ± 0.1
HHDG2	67.6 ± 0.5	66.6 ± 0.4	223.5 ± 0.2	68.0 ± 0.1	527.6 ± 0.4	68.3 ± 0.1

Note: Each compute node is equipped with 2 Tesla P100 GPUs. The four compute node setup has a peak performance of 74.4 TFLOP/s.

The Enki cluster [18] at NIST has thirteen compute nodes. Each node consists of two IBM Power9 CPUs, supporting 20 cores and 40 threads each, and 512GB of RAM.

Each node is equipped with 4 V100 Volta GPUs, each having a peak performance of 15.7 TFLOP/s for single precision. The code was compiled using gcc/11.2.1, OpenMPI/4.1.4 and cuda/11.7. Table 4 shows the performance results only for HHDG2 for four nodes, six nodes, and nine node configurations with varying matrix sizes. For different matrix sizes, the four-node configuration showed about 73.7% to 86% of peak performance, the six-node configuration showed about 68.4% to 79.2% of peak performance, while lastly, the nine-node configuration showed about 67% to 78.7% of peak performance. For nine node configurations, the experiments were conducted for limited matrix sizes due to time constraints. Also, we did not include the DPLASMA results because we ran into some system errors while working with this code on the Enki cluster.

Table 4: Enki

	4 Node (16 GPUs)		6 Nodes (24 GPUs)		9 Nodes (36 GPUs)	
N=M=K	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s	Time (sec)	TFLOP/s
100K	11.6 ± 0.1	185.1 ± 1.6	-	-	-	-
128K	21.1 ± 0.6	213.4 ± 5.5	15.3 ± 0.4	294.2 ± 7.3	-	-
150K	37.6 ± 0.8	192.7 ± 4.0	28.1 ± 0.8	257.9 ± 7.5	-	-
200K	90.5 ± 0.3	189.8 ± 0.5	65.1 ± 1.3	264.1 ± 5.1	-	-
250K	158.5 ± 0.7	211.7 ± 1.0	124.2 ± 1.1	270.2 ± 2.4	-	-
256K	166.5 ± 1.2	216.3 ± 1.5	127.6 ± 1.8	282.4 ± 4.2	95.7 ± 1.0	378.7 ± 4.1
300K	-	-	208.9 ± 2.2	277.6 ± 3.0	-	-
350K	-	-	311.5 ± 0.7	295.6 ± 0.7	-	-
400K	-	-	488.2 ± 0.9	281.5 ± 0.5	-	-
450K	-	-	656.0 ± 0.4	298.3 ± 0.2	-	-
512K	-	-	-	-	647.6 ± 4.0	445.1 ± 2.7

Note: Each compute node is equipped with 4 Tesla V100 GPUs. The peak performance for four-node, six-node and nine-node configurations is 251.2 TFLOP/s, 376.8 TFLOP/s and 565.2 TFLOP/s, respectively.

6 Conclusions and Future Work

This work aims to extend Hedgehog's abstractions while maintaining its programming model to operate in a cluster environment. Using matrix multiplication application as the vehicle, we were able to show that Hedgehog, with the new abstractions, performs on par with DPLASMA. These results show the efficiency and speed of the Hedgehog software as an engine for heterogeneous GPU computations.

The extension of Hedgehog to multiple nodes has been accomplished in a relatively straightforward fashion. The specialized *Sender* and *Receiver*, *DataWarehouse* tasks help provide a communication model that aligns with Hedgehog's out-of-order design while remaining agnostic of any particular communication framework like MPI. Also, the *Job* abstraction helps decouple the data distribution with the design of the data flow graph while providing a flexible way of defining workload distribution.

While the two GEMM implementations are very close, the key difference is that HHDG2 uses Hedgehog while DPLASMA uses ParSEC. The performance results of

HHDG2 demonstrates that Hedgehog provides a suitable mechanism for achieving high performance on single and multiple compute nodes.

Finally while the importance of the new abstractions described here is to show that Hedgehog's high performance may be achieved on a standard linear algebra benchmark, the next test will be to apply these ideas to a substantial engineering code and to show that the abstractions work equally well in this case. This is work that is ongoing.

Disclaimer

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification is not intended to imply recommendation or endorsement of any product or service by NIST, nor is it intended to imply that the materials or equipment identified are necessarily the best available for the purpose.

Code Availability

Our code is available here https://github.com/nitishingde/hh3-matmul-demo. The application v3_benchmark2 of the commit tagged as also v3_benchmark2 was used for benchmarking on all the systems.

References

[1] Shingde, N., Berzins, M., Blattner, T., Keyrouz, W., & Bardakoff, A. (2023). Extending Hedgehog's dataflow graphs to multi-node GPU architectures. In Lecture Notes in Computer Science (pp. 1–12). 10.1007/978-3-031-32316-4_1

[2] A. Bardakoff, B. Bachelet, T. Blattner, W. Keyrouz, G. C. Kroiz and L. Yon, "Hedgehog: Understandable Scheduler-Free Heterogeneous Asynchronous Multithreaded Data-Flow Graphs," 2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM), 2020, pp. 1-15., 10.1109/PAWATM51920.2020.00006.

[3] T. Herault, Y. Robert, G. Bosilca and J. Dongarra, "Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC," 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), 2019, pp. 33-41, 10.1109/ScalA49573.2019.00010.

[4] M. Gates, J. Kurzak, A. Charara, A. YarKhan, and J. Dongarra. 2019. SLATE: design of a modern distributed and accelerated linear algebra library. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19). Association for Computing Machinery, New York, NY, USA, Article 26, 1–18. 10.1145/3295500.3356223

[5] M Bauer, S Treichler, E Slaughter, and A. Aiken. 2012. Legion: Expressing locality and independence with logical regions. In Proc. of the Int. Conf. on High Perf. Comput., Networking, Storage and Analysis. IEEE Computer Society Press, 66.

[6] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight. 2016. Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices. SIAM Journal on Scientific Computing 38, 5 (2016), 101–122.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. Computing in Science Engineering 15, 6 (Nov 2013), 36–45.

[8] H. C. Edwards, C. R. Trott, and D. Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. J. Parallel and Distrib. Comput. 74, 12 (2014), 3202 – 3216.

[9] J. K. Holmen, D. Sahasrabudhe, M. Berzins. "A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems," In Proceedings of the Practice and Experience in Advanced Research Computing 2021 on Sustainability, Success and Impact (PEARC21), ACM, 2021

[10] J. K. Holmen, B. Peterson, M. Berzins. "An Approach for Indirectly Adopting a Performance Portability Layer in Large Legacy Codes," In 2nd International Workshop on Performance, Portability, and Productivity in HPC (P3HPC), SC19, 2019.

[11] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (Eugene, OR, USA) (PGAS '14). ACM, New York, NY, USA, Article 6.

[12] L. V Kale and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (Washington, D.C., USA) (OOPSLA '93). ACM, New York, NY, USA, 91–108.

[13] Q. Meng, A. Humphrey, M. Berzins. "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System," In Digital Proceedings of The International Conference for High Performance Computing, Networking, Storage and Analysis, SC'12, WOLFHPC 2012 Worshop, pp. 2441–2448. 2012.

[14] J.K. Holmen, D. Sahasrabudhe, M. Berzins. "Porting Uintah to Heterogeneous Systems," In Proceedings of the Platform for Advanced Scientific Computing Conference (PASC22) Best Paper Award, ACM, 2022.

[15] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009, 23:187-198, February 2011.

[16] Blumofe, R. D., & Leiserson, C. E. (1998). Space-Efficient Scheduling of Multithreaded Computations. SIAM Journal on Computing, 27(1), 202–229.

[17] Alexandre Bardakoff. Analysis and Execution of a Data-Flow Graph Explicit Model Using Static Metaprogramming. Université Clermont Auvergne, 2021. https://theses.hal.science/tel-03813645

[18] Computation Platform for AI/ML — NIST. (2019b, December 17). NIST. https://www.nist.gov/programs-projects/computation-platform-aiml

[19] Center for High Performance Computing - the University of Utah. (n.d.). https://chpc.utah.edu/

[20] Kaiser et al., (2020). HPX - The C++ Standard Library for Parallelism and Concurrency. Journal of Open Source Software, 5(53), 2352, https://doi.org/10.21105/joss.02352

[21] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pages 1–11. Supercomputing, IEEE.

[22] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198.

[23] M. Garland et al., "Parallel Computing Experiences with CUDA," in IEEE Micro, vol. 28, no. 4, pp. 13-27, July-Aug. 2008, doi: 10.1109/MM.2008.57. keywords: Parallel processing;Programming profession;Parallel programming;Concurrent computing;Computer architecture;Computer graphics;Kernel;Throughput;Central Processing Unit,

[24] Kale, L. V. and Krishnan, S. (1993). Charm++: A portable concurrent object oriented system based on c++. SIGPLAN Notices, 28(10):91–108.

[25] Bennett, J., Clay, R., Baker, G., Gamell, M., Hollman, D., Knight, S., Kolla, H., Sjaardema, G., Slattengren, N., Teranishi, K., et al. (2015). Asc atdm level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms. Technical Report SAND2015-8312, US Department of Energy, Sandia National Laboratories

[26] Alperen, Abdullah, Md Afibuzzaman, Fazlay Rabbi, M. Yusuf Ozkaya, Umit Catalyurek, and Hasan Metin Aktulga. “An Evaluation of Task-Parallel Frameworks for Sparse Solvers on Multicore and Manycore CPU Architectures.” In 50th International Conference on Parallel Processing, 1–11. Lemont IL USA: ACM, 2021. https://doi.org/10.1145/3472456.3472476.

[27] Gu, Ruidong, and Michela Becchi. “A Comparative Study of Parallel Programming Frameworks for Distributed GPU Applications.” In Proceedings of the 16th ACM International Conference on Computing Frontiers, 268–73. CF ’19. New York, NY, USA: Association for Computing Machinery, 2019. https://doi.org/10.1145/3310273.3323071.

[28] Agullo, Emmanuel, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. “Task-Based Parallel Programming for Scalable Matrix Product Algorithms.” ACM Transactions on Mathematical Software 49, no. 2 (June 30, 2023): 1–23. https://doi.org/10.1145/3583560.

[29] Rohr, David, and Volker Lindenstruth. “A Flexible and Portable Large-Scale DGEMM Library for Linpack on Next-Generation Multi-GPU Systems.” In 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 664–68, 2015. https://doi.org/10.1109/PDP.2015.89.

[30] Baker, Gavin Matthew, Bettencourt, Matthew Tyler, Bova, Steven W., Franko, Ken, Gamell, Marc, Grant, Ryan, Hammond, Simon David, Hollman, David S., Knight, Samuel, Kolla, Hemanth, Lin, Paul, Olivier, Stephen Lecler, Sjaardema, Gregory D., Slattengren, Nicole Lemaster, Teranishi, Keita, Wilke, Jeremiah J., Bennett, Janine Camille, Clay, Robert L., Kale, Laxkimant, Jain, Nikhil, Mikida, Eric, Aiken, Alex, Bauer, Michael, Lee, Wonchan, Slaughter, Elliott, Treichler, Sean, Berzins, Martin, Harman, Todd, Humphreys, Alan, Schmidt, John, Sunderland, Dan, Mccormick, Pat, Gutierrez, Samuel, Shulz, Martin, Gamblin, Todd, and Bremer, Peer -Timo. 2015. ”ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms”. United States. https://doi.org/10.2172/1432926. https://www.osti.gov/servlets/purl/1432926.

[31] Wu, Nanmiao, Ioannis Gonidelis, Simeng Liu, Zane Fink, Nikunj Gupta, Karame Mohammadiporshokooh, Patrick Diehl, Hartmut Kaiser, and Laxmikant V. Kale. “Quantifying Overheads in Charm++ and HPX Using Task Bench.” In Euro-Par 2022: Parallel Processing Workshops, edited by Jeremy Singer, Yehia Elkhatib, Dora Blanco Heras, Patrick Diehl, Nick Brown, and Aleksandar Ilic, 5–16. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023. https://doi.org/10.1007/978-3-031-31209-0_1.