Extending Hedgehog's dataflow graphs to multi-node GPU architectures[*]

Nitish Shingde[1][0000−0002−2616−7971], Martin Berzins[1][0000−0002−5419−0634], Timothy Blattner[2][0000−0002−4964−5403], Walid Keyrouz[2][0000−0003−3807−813X], and Alexandre Bardakoff[2][0000−0003−2770−8052]

[1] University of Utah, Utah SLC 84112, USA
{nitish,mb}@sci.utah.edu
[2] National Institute of Standards & Technology, Gaithersburg MD, USA
{timothy.blattner,walid.keyrouz}@nist.gov
a.bardakoff@prometheuscomputing.com

Abstract. Asynchronous task-based systems offer the possibility of making it easier to take advantage of scalable heterogeneous architectures. This paper extends the National Institute of Standards and Technology's Hedgehog dataflow graph models, which target a single high-end compute node, to run on a cluster by borrowing aspects of Uintah's cluster-scale task graphs and applying them to a sample implementation of matrix multiplication. These results are compared to implementations using the leading libraries, SLATE and DPLASMA, for illustrative purposes only. The motivation behind this work is to demonstrate that using general purpose high-level abstractions, such as Hedgehog's dataflow graphs, does not negatively impact performance.

Keywords: Hedgehog · multi-node GPU · dataflow · task graphs · Uintah · MPI

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 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 [1] and the Uintah Computational Framework [7,16]. Hedgehog specializes in node-level performance and uses C++ threads and NVIDIA CUDA. Uintah specializes in large-scale simulations and uses an MPI+X hybrid parallelism model. Both systems use asynchronous execution to achieve. This paper shows that Hedgehog may be extended by making use of the general philosophy of Uintah. It compares the performance that may be achieved with a prototype version against the well-known DPLASMA and SLATE frameworks. This work builds on the prior work

[*] Supported by organization NIST.

of Holeman [10]; the vehicle for comparison is the well-studied problem of dense matrix-matrix multiplication.

Matrix multiplication performance has improved greatly with the advancement of accelerated devices. Two of the best-known libraries out there that use out-of-core matrix multiplication on multi-GPU accelerated nodes are DPLASMA[5] and SLATE[3]. DPLASMA provides a generic and flexible matrix-matrix multiplication algorithm C = A×B for multi-GPU accelerated distributed-memory platforms for matrices unrestricted by the size of the GPU memory. The implementation relies on the classical tile-based outer-product algorithm but enhances it with several control dependencies to increase data reuse and optimize communication flow from/to the accelerators within each node. The implementation uses the Parsec runtime system, another task-based runtime system. SLATE is designed to deliver fundamental dense linear algebra capabilities for current and upcoming distributed-memory systems. It is built on top of standards, such as MPI and OpenMP, and de-facto industry solutions, such as NVIDIA CUDA and AMD HIP.

The rest of the paper is organized as follows. Section 2 discusses the various frameworks that deal with multi-GPU distributed-memory platforms. With the matrix multiplication problem as a vehicle, the section discusses how some existing state-of-the-art techniques tackle the situation. Section 3 presents the design principles used to implement matrix multiplication for a multi-GPU accelerated distributed-memory platform. Section 4 discusses the design principles of matrix multiplication using Hedgehog. After describing Hedgehog's single-node multi-GPU solution, the extension to multiple nodes will be given. Section 5 compares Hedgehog's results against those of SLATE and DPLASMA. Section 6 concludes the paper leaving section 7 with possible future plans.

2 Existing approaches

2.1 Uintah

Part of the original motivation for the extension of the Hedgehog system to multiple 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 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++ [14], HPX [13], Legion [6], PaRSEC [8], and Uintah [7].

While Uintah has demonstrated large scale scalability on heterogeneous architectures [16], it started as a fixed task-graph execution code and was extended to dynamic task execution [15]. 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 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 [9], 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 [12], 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 [11] and the accompanying portable abstractions [12] to execute workloads representative of typical Uintah applications. The recent results in [16] at scale shows 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 to prioritize external communication while hiding its impact using overdecomposition.

2.2 DPLASMA

DPLASMA is a distributed parallel linear algebra software targeted toward multicore architectures. The matrix multiplication algorithm uses the Parameterized Task Grap (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 node. It uses cuBLAS's General Matrix Multiplication (GEMM) kernel for computation and MPI for nodal communication.

2.3 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].

2.4 Hedgehog

Hedgehog [20] 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 only managed by the operating systems, and execute based on the presence of data.

The Hedgehog nodes are attached with edges representing the flow of data using queues that store unprocessed data. The nodes and edges are structured under 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 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, which share the same input and output edges consisting of queues and synchronization contexts. State manager tasks use localized state, which are 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-node computations in [1]. 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.

3 Extending Hedgehog to Multiple 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 multi-node scaling with two new specialty tasks: (1) *Sender* and (2) *Receiver* tasks. Similarly to Uintah [7], each node has its own local task graph instead of having a global task graph to manage work across the nodes for scalability. Each of these local graphs contains these two new specialty tasks to establish a form of communication. In Section 4 the *Sender* and *Receiver* tasks are implemented for matrix multiplication and deal with point-to-point communication. Though these tasks use MPI underneath as their communication framework, they are designed to be agnostic of such communication models. In the following section, the term *data* will signify data that flow within a local Hedgehog task graph, whereas the term *message* will represent the data that travel across nodes.

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/nodes. The incoming data to the sender task specify the destination 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 node for the Sender task to pack the data into a DataPacket and send it across to the *Receiver* task of the receiving 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. 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 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. This would allow for the receiving thread to enter into a wait state until a message is incoming.

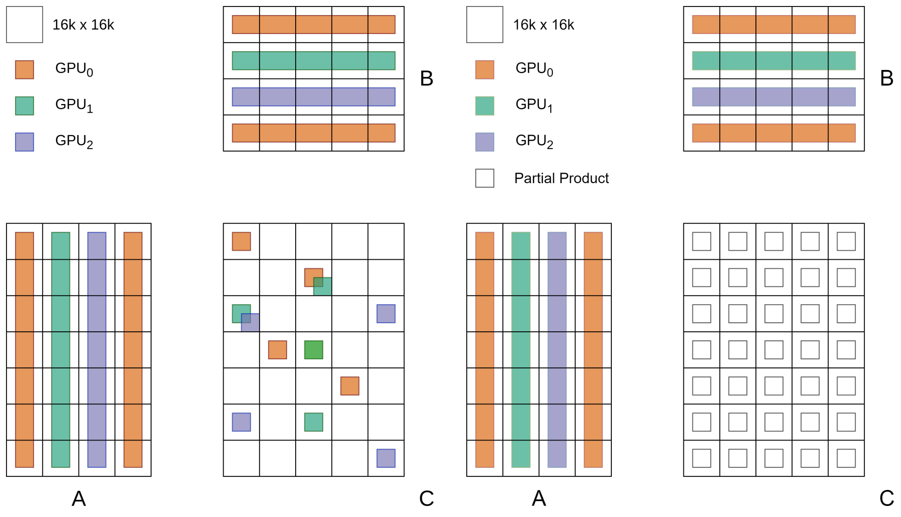
4 Matrix Multiplication using Hedgehog

The algorithm implemented here is an extension of the single node setup implemented in section 4.3 of Alexandre's thesis [20]. The thesis explores the algorithm's evolution from CPU only to CPU+GPU to CPU+multiple GPUs using Hedgehog. We briefly revisit the single-node setup and then its subsequent evolution to multiple nodes using the abstractions mentioned in section 3. While the approach used here lays down the general approach to extend Hedgehog to multiple 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, N_T, and K_T represent the number of tiles along the M, N, and K dimensions of the matrices, respectively.

4.1 Single-node setup

Figure 1 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.



(a) Data Distribution (b) Work Distribution

Fig. 1: 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 [20].

In Hedgehog, the task graph is instantiated only once during its creation. When a task receives new data, the data simply wait in a queue if all the threads concerning the tasks are busy. This differs from traditionally used task graphs in systems like StarPU [18], PLASMA, and CILK [19], 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.

4.2 Multiple node setup

Figure 2 highlights the data distribution in a multi-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-node setup to independently compute partial results for every element in matrix C. In the current design, every node calculates a partial result for all the elements in matrix C. We need to reduce the matrix C present on each node to get the final result. There are two types of accumulations happening here, one within a 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 nodes increase. The accumulation of matrix C tiles (within a 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 2 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 helped 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.

4.3 Communications

As discussed above, no inter-node communication occurs for matrices A and B. The only communication that takes place is for matrix C. Even with the above asynchronous approach for reducing the matrix C, the communication volume is equivalent to a collective reduction call, which is $M_T * N_T * (n-1)$ number of tiles, where n is the number of nodes.

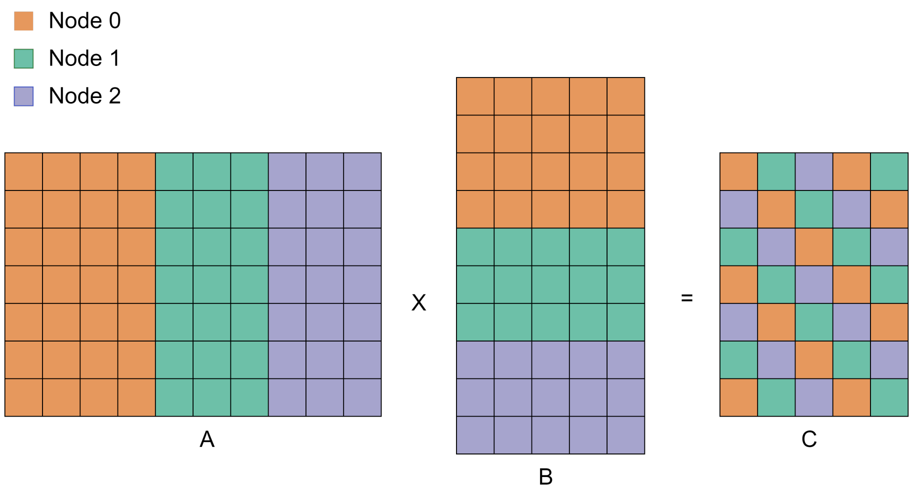


Fig. 2: Data distribution of matrices across multiple nodes. Matrices A and B are distributed in a 1D Block Column and 1D Block Row fashion respectively. Matrix C, as a whole, redundantly resides all the nodes with the ownership marked in 2D Block cyclic fashion.

Within a node, while copying the matrix data to GPU memory, only tiles from matrices A and B are transferred, and that too only once. In total, for a given node, all the $M_T * \frac{K_T}{n}$ tiles from submatrix A, and $N_T * \frac{K_T}{n}$ tiles from submatrix 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 $M_T * N_T * \frac{K_T}{n}$ times. Therefore, the total communication volume, in terms of tiles, to and fro per node is $\frac{K_T}{n}(M_T * N_T + M_T + N_T)$.

5 Results

All the experiments were conducted at CHPC, the Center for High-Performance Computing at the University of Utah. We picked 6 compute nodes, each containing a 64-core AMD third-generation (Milan) 7713P processor. Two nodes consisted of 8 Nvidia RTX A6000 48GB GPUs per node; the other two nodes consisted of 2 Nvidia A100 80 GB PCIe GPUs per node, and the remaining two nodes consisted of 8 Nvidia A100 80GB SXM4 per node. The first four nodes had 256GB of CPU memory, while the last two had 512 GB. Each node had connectX6 HDR Infiniband cards connected with EDR Infiniband. For the 4-node experiment, 3 GPUs per node were used, and for the 6-node experiment, 2 GPUs per node. Both experiments used 12 GPUs in total. Every run is measured over ten iterations and presented as mean and standard deviations of the execution times (seconds) and performances (TFLOP/s).

Tables 1 and 2 show the performance results for single precision square matrices for A, B, and C of length 192K with different node configurations. The best tile size was selected for DPLASMA and SLATE based on all runs with variable tile sizes on our nodes. The Hedgehog implementation performs on par with DPLASMA and SLATE on both 4-node and 6-node experiments. The double precision experiments were not conducted due to the lack of accelerated double precision performance in the Nvidia RTX A6000 GPUs.

Table 1: Mean and standard deviation of run times and performance on 4 nodes, with 3 GPUs per node configuration, using single precision 192K x 192K matrices.

Algo	Time (seconds)	TFLOP/s
DPLASMA	85.23 ± 0.90	178.34 ± 1.87
SLATE	82.74 ± 0.18	183.70 ± 0.42
Hedgehog	85.86 ± 1.89	177.09 ± 3.87

Table 2: Mean and standard deviation of run times and performance on 6 nodes, with 2 GPUs per node configuration, using single precision 192K x 192K matrices.

Algo	Time (seconds)	TFLOP/s
DPLASMA	85.52 ± 0.55	177.73 ± 1.15
SLATE	82.31 ± 0.21	184.64 ± 0.49
Hedgehog	85.92 ± 2.44	177.01 ± 4.77

6 Conclusions

This work aims to extend Hedgehog's abstractions while maintaining its programming model to operate in a cluster environment. We have shown that it is possible to obtain performance in multi-node Hedgehog that is on par with well-known libraries.

The extension of Hedgehog to multiple nodes has been accomplished in a relatively straightforward fashion. The specialized *Sender* and *Receiver* 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.

There are some caveats with the current approach for matrix multiplication, as it is not yet fully scalable because of redundant copies of matrix C on every node. This implementation also fails to apply proper load balancing for oversubscribed GPUs. The DPLASMA and SLATE libraries outperformed Hedgehog by a margin of 30 % and 20 %, respectively, when more GPUs were allocated for the same matrix configuration. However, the results are a good starting point for the proposed future work using Hedgehog on more general parallel computing examples.

7 Future work

The matrix multiplication algorithm could also be tackled by partitioning the work, i.e., focusing on matrix C part by part, like a sliding window. This technique also provides the flexibility to accommodate the possible limitations of GPU and host memory, but at the cost of increased intra-node communication.

One important next step to this work is to add two abstractions to generalize the approach; first, a serialization/deserialization abstraction to our Sender and Receiver task to help deal with complicated data structures; and second, an abstraction for defining decomposition strategies, which can be used to automatically determine where data reside across nodes.

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.

References

1. 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. , https://doi.org/10.1109/PAWATM51920.2020.00006.
2. 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, https://doi.org/10.1109/ScalA49573.2019.00010.
3. Kurzak, J., Gates, M., Charara, A., YarKhan, A., Yamazaki, I., Dongarra, J. (2019). Linear Systems Solvers for Distributed-Memory Machines with GPU Accelerators. In: Yahyapour, R. (eds) Euro-Par 2019: Parallel Processing. Euro-Par 2019. Lecture Notes in Computer Science(), vol 11725. Springer, Cham. https://doi.org/10.1007/978-3-030-29400-7_35
4. Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack 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. https://doi.org/10.1145/3295500.3356223
5. G. Bosilca et al., "Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA," 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 1432-1441, doi: 10.1109/IPDPS.2011.299.
6. 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.

7. 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.
8. 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.
9. 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.
10. J. K. Holmen, D. Sahasrabudhe, M. Berzins, A. Bardakoff, T. J. Blattner, . Keyrouz. “Uintah+Hedgehog: Combining Parallelism Models for End-to-End Large-Scale Simulation Performance,” Scientific Computing and Imaging Institute, 2021.
11. 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
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. D. Vandevoorde, N.M Josuttis and D. Gregor, ”C++ Templates: The Complete Guide (2nd Ed.)”, Addison-Wesley Professional, 2017, ISBN 0321714121.
18. Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André 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.
19. Blumofe, R. D., & Leiserson, C. E. (1998). Space-Efficient Scheduling of Multithreaded Computations. SIAM Journal on Computing, 27(1), 202–229.
20. 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`