Hedgehog: Understandable Scheduler-Free Heterogeneous Asynchronous Multithreaded Data-Flow Graphs

Alexandre Bardakoff, Bruno Bachelet, Timothy Blattner, Walid Keyrouz, Gerson C. Kroiz, Loïc Yon

Hedgehog: Understandable Scheduler-Free Heterogeneous Asynchronous Multithreaded Data-Flow Graphs

Alexandre Bardakoff[1,2], Bruno Bachelet[2], Timothy Blattner[1], Walid Keyrouz[1], Gerson C. Kroiz[4], and Loïc Yon[3]

[1]National Institute of Standards & Technology, Gaithersburg, MD 20899-8970, email: first.last@nist.gov
[2]Université Clermont Auvergne, CNRS, LIMOS, F-63000 Clermont-Ferrand, France, email: first.last@uca.fr
[3]ISIMA, CNRS, LIMOS, F-63000 Clermont-Ferrand, France, email: first.last@isima.fr
[4]Department of Mathematics and Statistics, University of Maryland, Baltimore County, Baltimore, MD 21250, USA

Abstract

Getting performance on high-end heterogeneous nodes is challenging. This is due to the large semantic gap between a computation's specification—possibly mathematical formulas or an abstract sequential algorithm—and its parallel implementation; this gap obscures the program's parallel structures and how it gains or loses performance. We present Hedgehog, a library aimed at coarse-grain parallelism. It explicitly embeds a data-flow graph in a program and uses this graph at runtime to drive the program's execution so it takes advantage of hardware parallelism (multicore CPUs and multiple accelerators). Hedgehog has asynchronicity built in. It statically binds individual threads to graph nodes, which are ready to fire when any of their inputs are available. This allows Hedgehog to avoid using a global scheduler and the loss of performance associated with global synchronizations and managing of thread pools. Hedgehog provides a separation of concerns and distinguishes between compute and state maintenance tasks. Its API reflects this separation and allows a developer to gain a better understanding of performance when executing the graph. Hedgehog is implemented as a C++ 17 headers-only library. One feature of the framework is its low overhead; it transfers control of data between two nodes in $\approx 1\,\mu s$. This low overhead combines with Hedgehog's API to provide essentially cost-free profiling of the graph, thereby enabling *experimentation for performance*, which enhances a developer's insight into a program's performance.

Hedgehog's asynchronous data-flow graph supports a data streaming programming model both within and between graphs. We demonstrate the effectiveness of this approach by highlighting the performance of *streaming* implementations of two numerical linear algebra routines, which are comparable to existing libraries: matrix multiplication achieves $>95\,\%$ of the theoretical peak of 4 GPUs; LU decomposition with partial pivoting starts streaming partial final result blocks $40\times$ earlier than waiting for the full result. The relative ease and understandability of obtaining performance with Hedgehog promises to enable non-specialists to target performance on high-end single nodes.

1 Introduction

Parallel programs are increasing in complexity as they target massively parallel platforms (e.g., 2×64 or more CPU cores and multiple GPUs). Obtaining performance on such platforms is challenging because of the large semantic gap between a computation's specification and its implementation. The specification is often as simple as a set of mathematical formulas or an abstract sequential algorithm. By contrast, a parallel implementation must address several coupled issues beyond what a sequential implementation addresses: concurrent computations, multiple memory

resources, race conditions and deadlocks, and data motion costs. One approach to bridge this gap is to use a framework that simplifies application development and, equally important, exposes *abstractions* that address parallelism and performance as first-class concerns. Furthermore, these abstractions should represent parallel constructs and make it easier to instrument and reason about an application's performance thereby allowing developers to gain deeper insight.

This paper presents Hedgehog, a general-purpose performance-oriented library aimed at coarse-grain parallelism on single high-end heterogeneous compute nodes. A Hedgehog program contains an explicit representation of a static data-flow graph. The graph executes in a pure asynchronous data-driven mode without a global scheduler and statically binds threads to persistent tasks in the graph. Hedgehog provides a *separation of concerns* and distinguishes between *compute* tasks (i.e., compute-bound kernels) and *state manager* tasks. State managers represent and manage *local* states between the compute tasks that they connect to. Hedgehog also provides a *memory manager* tool to control the use of memory resources within a task.

Hedgehog transfers control of data efficiently between two tasks ($\approx 1\,\mu s$). Its explicit data-flow representation and the graph's cost-free profiling allow the developer to quickly identify how the overall computation is carried out. In addition, it encourages a developer to experiment with the data-flow graph itself and with how to customize the degree of parallelism of compute tasks.

The rest of this paper is organized as follows. Section 2 presents solution requirements. Section 3 reviews existing approaches and frameworks. Section 4 discusses Hedgehog and provides a high-level view of its implementation. Section 5 examines two linear algebra examples using Hedgehog and highlights obtained performances. Section 6 concludes and Section 7 outlines future work.

2 Solution Approach

Developing an application to scale on a high-end heterogeneous node is challenging. These nodes have a high degree of hardware parallelism with high core-count CPUs (currently $2\times$ 64-core CPUs, 128-core CPUs soon), multiple accelerators and GPUs (up to 4 or 8 per node), and multi-instance GPUs ($7\times$ per Nvidia A100 GPU). Furthermore, these nodes have become commodity and, as such, are now accessible to a broad community that is much disconnected from the traditional HPC community and lacks its deep knowledge, which is needed to take advantage of these nodes. An enabling solution for this nascent HPC community should compose optimized compute kernels and efficient data transfers between memory domains (CPU and GPU), express the locality of data, overlap data motion and computations, and fully utilize available hardware. Equally important, this solution should have an explicit and understandable program and execution model that is accessible to both expert and non-expert developers so they can improve their code's performance. This model will allow developers to operate at higher levels of abstractions and will also make it easier for them to adapt their software designs to future architectures as hardware evolves.

Existing approaches can exploit the compute power available in high-end nodes, but do so with an implicit execution model that requires the use of external profiling tools to reveal the impact of software design decisions on performance. This makes it challenging for developers to reason about optimization strategies. Furthermore, these external tools may have substantial overhead; this may make it even more challenging for developers to pinpoint performance bottlenecks.

Our solution, Hedgehog, is based on a static and explicit data-flow graph that operates without a global scheduler and is aimed at coarse-grain parallelism. The execution model obtains performance via *data pipelining* strategies, which works best with streaming data. Using this approach, the developer can effectively overlap data motion and computations to keep the hardware busy. The model is accompanied by tools that assist with memory management to express data locality in multiple memory domains (CPU and GPUs), thereby addressing memory constraints on a heterogeneous node. We use the explicit representation throughout the execution and profiling of the program; this leads to cost-free visual feedback. The representation and feedback make it easier to reason about the graph's execution and encourage the developer to experiment with the graph so as to optimize its performance. This experimentation can lead to quickly identifying the

performance critical path in an implementation's graph. We designate this approach as *portable designs for performance* because the underlying data-flow graph can be easily tuned to different high-end nodes. These decisions can be analyzed in conjunction with the graph profiling to identify optimal configurations. This iterative optimization approach also applies to improving tasks to take better advantage of hardware, also called *experimentation for performance*.

3 Background/State of the Art

Multiple models exist for developing parallel applications that maximize the utilization of available hardware and can target future heterogeneous nodes. The three most commonly used approaches are parallel libraries, language extensions, and task-based libraries.

Parallelized libraries, such as OpenBLAS [22], OpenCV [6], or FFTW [8], implement parallelism within each function call and effectively establish a *synchronization barrier* at the end of each function invocation in the parallelized library. Hedgehog encourages the use of parallelized libraries in single-threaded mode within its tasks for their optimized implementations that use hardware vector instructions for example.

Directive-based approaches, such as OpenMP [18], OpenACC [10], and OmpSs-2 [16] use mostly pragmas or codelets. These approaches focus on using loop parallelism to obtain performance. They offer many options to customize the pragmas, which can become very complex when handling loops with multiple dependencies and synchronization points. Therefore, to be used correctly, they often require power users with a deep understanding of the hardware and have knowledge about such parallel programming techniques. The data-flow model is capable of fully utilizing a high-end node by orchestrating coarse-grain task parallelism. It also allows an end-user to invoke parallel directive-based kernels from existing libraries in a task.

Task-based approaches exist under numerous forms with different properties. The majority of task-based libraries operate with two prevalent traits: (1) the graph's representation as a DAG (Directed Acyclic Graph) and (2) the usage of a pool of threads. The programmer describes the DAG by specifying dependencies between tasks, which task is applicable to either a processor, co-processor, or both. Efficiently binding these tasks to threads in the thread pool, which also selects either the processor or co-processor as the compute resource, is non-trivial. There is no perfect dynamic matching algorithm, which is considered a NP-complete problem, but optimizers can be used to improve the matching. Furthermore, to speed up the overall computation, many task-based approaches add work-stealing or work-balancing techniques at the cost of increasing overhead. These steps define how the library intrinsically works, and are hidden from the algorithm developer in the interest of simplifying parallel programming.

Existing task-based libraries, such as HPX [11], Legion [2], StarPU [1] or Charm++ [12], target performance at any scale from fine-grained parallelism to distributed parallel computing. HPX uses an API compatible with the C++ 14 standard and is based on message-driven computations and asynchronous calls. Legion bases its computation representation on the decomposition of its data. The logical regions express locality and dependence of program data and tasks that act on the logical regions. StarPU uses a codelet approach. Charm++ is a message-driven library that depends on asynchronous calls. Hedgehog focuses on maximizing hardware utilization on a single node, while maintaining low latency, from 1 µs to 10 µs for a large variety of tasks. Tasks are represented as C++ classes that embed useful information for the computation (class attributes, static variables, usage of input/output streams) as opposed to using codelets.

Specialized libraries, such as Uintah [15], Kokkos [4], or Halide [19], have been developed with key design decisions for specific application classes. Uintah is based on adaptive mesh refinement with a runtime system aimed at solving partial differential equations in large scale simulations. Kokkos, a library used for manycore parallelism on clusters with MPI, targets HPC applications. It also comes with a co-library, kokkos-kernel, specialized for linear algebra. Halide, a language embedded in C++, targets image processing algorithms. Hedgehog has no specialization, but benefits most from an effective coarse-grained data decomposition that feeds sufficient data into the data-flow graph.

Intel Threading Building Blocks [14] is a template library for parallel programming. It proposes an algorithm as skeletons along with parallel containers to achieve performance on general algorithms. An algorithm is automatically represented internally as a graph with the library in charge of managing the algorithm's graph from creation to tear-down with work stealing for load-balancing on the node. Hedgehog uses an explicit graph to construct the algorithm and maintains it during execution. No scheduling or balancing are used because Hedgehog relies only on its data-flow representation.

Hedgehog evolved from *HTGS* [3]; they are a part of task-based approach with some major differences explained in Section 4. Hedgehog is at least as efficient as *HTGS*, according to a study conducted in Section 5.1. There is no loss of features; instead some features have been added such as multiple inputs, broadcast, and an internal re-architecture transforming the HTGS *Bookkeeper* into the *State Manager*.

4 Hedgehog

Hedgehog is a software infrastructure that presents an algorithm at a high level of abstraction and helps developers reason about their applications. The library is header-only and implemented using the C++ 17 standard, and requires no other external dependencies.

Algorithms are formulated into a data-flow graph using the Hedgehog library. Nodes in the graph are persistent entities that accept and produce data. Edges connect nodes using queues that store data.

Nodes in Hedgehog have been designed for non-overlapping usage to achieve a separation of concerns. A task is a node that carries out computations, where each task is statically bound to a thread. The state manager is a specialized task that embeds a shareable thread-safe state object to locally manage and do synchronization on the computation flow. This separation of concerns is a cornerstone of Hedgehog's design and facilitates the programmability of the library. The nodes and edges operate without any added scheduler and, alongside the threads attached to each task, formulate an asynchronous data pipeline, which allows Hedgehog to overlap computation, data motion, and I/O. It leverages streaming execution using data decomposition to maximize the system utilization.

The graph is also a node and can be embedded in another graph; this structure enables composition and code sharing. The Hedgehog graph is static so there is no back-end reorganization or expansion of the graph.

To present Hedgehog and its model, we first present the data-flow model in Section 4.1, with the different kinds of nodes in Section 4.2 and their interfaces in Section 4.3. Section 4.4 uncovers our threading model and how the computation is conducted without a scheduler. Section 4.5 and Section 4.6 reveal tools embedded in Hedgehog such as the memory manager, and our cost-free visual feedback. Section 4.7 exposes how Hedgehog enforces type compatibility with C++ metaprogramming techniques.

To illustrate the various components of the Hedgehog library, we present a CPU-based implementation of matrix multiplication. In Section 5, we present the results of a variant of this implementation that targets multi-GPU nodes to showcase the performance and low overhead of Hedgehog's operation. To be clear, Hedgehog is a general purpose library; its programming model has been used in a variety of applications such as image processing and signal processing.

In this example of matrix multiplication, we will create a BLAS-like routine, similar to GEneral Matrix Multiplication (*GEMM*), $C = A \times B + C$ with matrices $A_{n \times m}$, $B_{m \times p}$, and $C_{n \times p}$. These matrices are decomposed into blocks: C is decomposed into $N \times P$ blocks along the rows and columns of the matrix, respectively. In order to access a block at row r and column c we use the notation $C_{r,c}$. The computation is achieved by iterating for each block in C, $C_{r,c} = \sum_{i=1}^{M}(A_{r,i} \times B_{i,c})$, where M is the number of blocks along the shared dimension. This example as well as others are available on github [9]. These tutorials demonstrate the API.

4.1 Structural model: data-flow graph

The first step to programming with Hedgehog is to understand how data flows within an algorithm. The best way to approach this step is to formulate the algorithm as a data-flow graph, which is a representation for computation using a directed graph. The graph's nodes are the computation actors and the graph's edges are the directed information flow. The graph has one entry point and one exit point.

Every task executes based on the presence of data in its input queues, which results in a pipeline that executes concurrently. This threading model is described with more detail in Section 4.4.

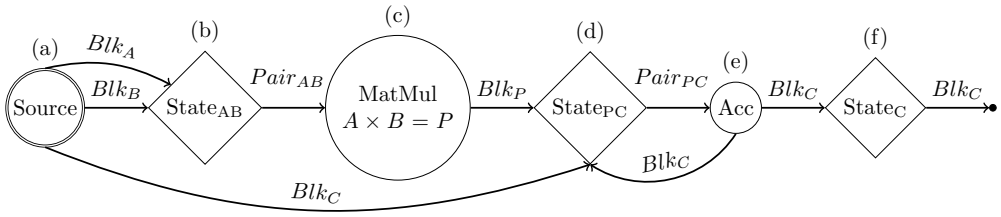

Figure 1: Matrix multiplication data-flow graph

Figure 1 shows the structure for the matrix multiplication data-flow graph, where circles are the tasks and diamonds are the state managers, can be split into the following computational steps:

1. The source (a) of the graph dispatches blocks of matrices A, B, and C to the input nodes.
2. The $State_{AB}$ (b) accepts blocks of A and B, groups them into compatible pairs of blocks, and emits pairs of compatible blocks.
3. The $MatMul$ (c) task accepts pairs of A and B blocks, applies matrix multiplication on them, and produces partial results P.
4. The $State_{PC}$ (d) produces a $Pair_{PC}$ output when it receives one Blk_C and one Blk_P that need to be accumulated.
5. The Acc (e) task accumulates the appropriate C blocks with the P blocks.
6. The $State_C$ (f) produces an output when Blk_C is fully accumulated.

```
using MatrixType = float;// Type of Matrix elements
Order Ord = Order::Column;// Matrix orientation
// Tasks
auto matMulTask =
  std::make_shared<MatMul<MatrixType, Ord>>(numberThreadProduct, p);
auto accTask =
  std::make_shared<AccTask<MatrixType, Ord>>(numberThreadAddition);
//[...]
// Build the graph
matMulGraph.input(stateAB);
matMulGraph.input(statePC);
matMulGraph.addEdge(stateAB,matMulTask);
matMulGraph.addEdge(matMulTask,statePC);
matMulGraph.addEdge(StatePC,accTask);
matMulGraph.addEdge(accTask,statePC);
matMulGraph.addEdge(accTask,stateC);
matMulGraph.output(stateC);
```

Listing 1: Simplified MatMul main

The Hedgehog *Graph* adds these tasks by establishing an edge between them with the method `graph.addEdge(SenderNode, ReceiverNode)`, which will automatically generate FIFO (First In First Out) data queues. Methods from the graph connect tasks to the inputs using `graph.input(InputNode)` and outputs using `graph.output(OutputNode)` as shown in Listing 1.

4.2 Hedgehog nodes

Hedgehog provides several distinct types of nodes that are used to develop data-flow graphs. The nodes are designed to provide specific functionality in order to achieve a separation of concerns approach; i.e., nodes that specifically target state maintenance, computation, composition, or scalability. Each node type is restricted to consuming multiple input types and producing a single output type, as described in Section 4.3. The node classes form a hierarchy to enable expanding upon the Hedgehog library, as shown in Figure 2. The available Hedgehog node types are (1) *Graph*, (2) *AbstractTask*, (3) *StateManager*, (4) *CUDATask*, and (5) *ExecutionPipeline*.

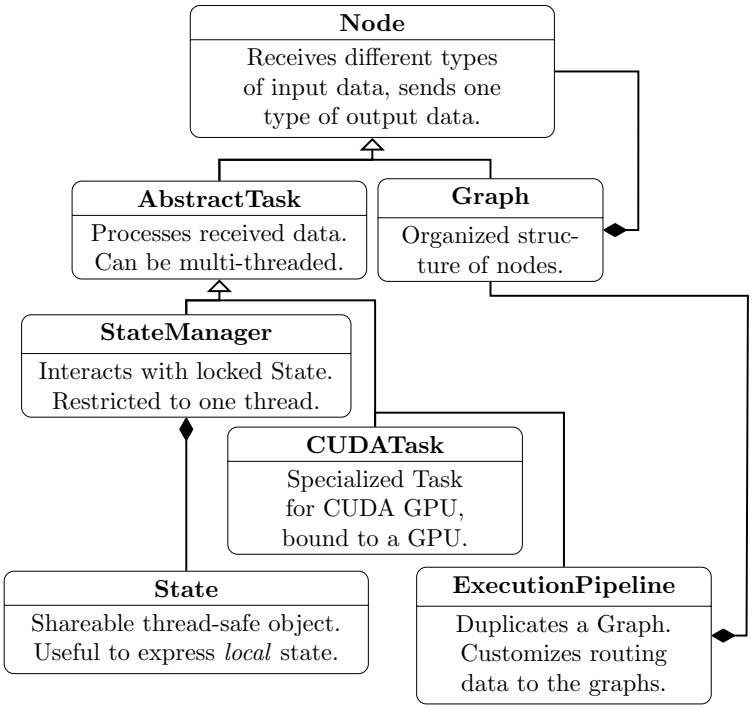

Figure 2: UML simplified class hierarchy of Hedgehog

Node is the pure abstract class at the root of the class hierarchy; it is the base class for all Hedgehog node objects. The *Graph* object represents a computation, it connects a set of consistent nodes and edges to carry out the computation. In Hedgehog, a node must be associated with a parent graph, which binds its constituent nodes to the same device to improve data locality.

The *AbstractTask* node operates on data; it is abstract as it does not have an implementation for the *execute* methods of all input types. *execute* method is called with data that is taken from the task's input queue(s). It executes a computational kernel on the data it receives as shown in Figure 3. This figure shows the execution logic of an *AbstractTask*, and displays the customization points. The *canTerminate* function identifies when to stop a task; by default, this is when there are no input nodes connected and nothing in the input queues. This behavior can be modified to break cycles in the graph. The *initialize* and *shutdown* functions are called once to carry out pre- or post- computation. Listing 2 for a simplified *MatMul* task, which implements the *execute* method to carry out the matrix multiplication operation.

```
class MatMul :
  public AbstractTask<BlkP, pair<BlkA, BlkB>>{
private:
  size_t count_ = 0;
public:
  MatMul(size_t nbThreads, size_t count)
  : AbstractTask<BlkP, pair<BlkA, BlkB>(
  "Product Task", nbThreads), count_(count){}

  void execute(pair<BlkA,BlkB> ptr){
    auto matA = ptr.first;
    auto matB = ptr.second;
    BlkP res {};
    // res <- matA*matB
    cblas_sgemm(matA, matB, res);
    this->addResult(res);
  }
};
```

Listing 2: Simplified version of MatMul task

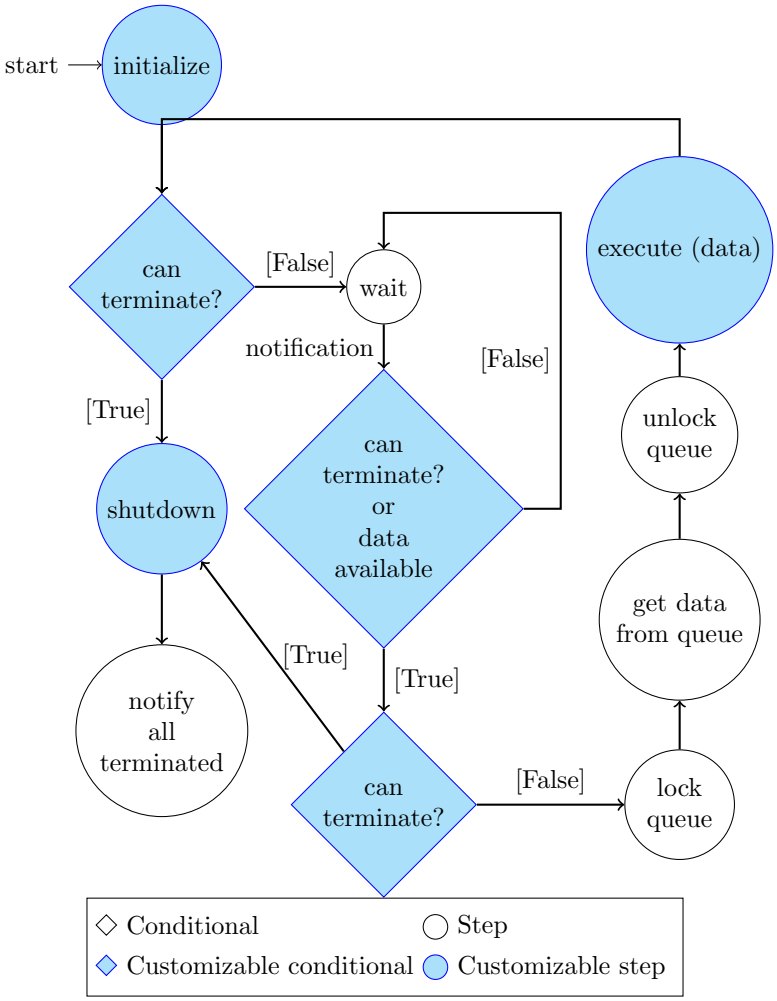

Figure 3: Task's main runtime steps

The first specialized *AbstractTask* is the *StateManager*. The *StateManager* holds a *State* instance. The *State* expresses a *localized* state between two or more nodes. Once an input data is available, the *StateManager* locks the *State* it owns and transfers the data to the *State*. Once the *State* computation is done, its output data are gathered by the *StateManager*, the *State* is unlocked and the gathered data are sent to its connected nodes.

The shareable *State* object contains a mutual exclusion context to ensure it is only accessed by one thread at a time. Both the *StateManager* and *State* are fully customizable, but operate with the distinct requirement that *StateManager* is the only type of task in the graph that should express the state of a computation.

The second specialized *AbstractTask* is the *CUDATask*. It binds the task's thread to a CUDA-enabled GPU, which is done automatically by Hedgehog through a customization of the *initialize* function. This is used to streamline executing a CUDA kernel, such as by automatically enabling CUDA peer-to-peer access and creating a *cudaStream* for each instance of the *CUDATask*.

The last specialized task in Hedgehog is the *ExecutionPipeline* which is used to simplify multi-GPU programming or other execution contexts. This task creates mirror copies of a graph and distributes data between graph copies according to a developer-specified strategy. Each graph copy is bound to a separate GPU; the graph's data will be local to its tasks and, as such, to the GPU that it binds to. When a *StateManager* is copied, its *State* instance is shared across all *StateManager* instances. This is often used to share data across multiple graphs, and potentially between GPUs. For example, data that is stored within the *State* can be retrieved and used by another graph from its *StateManager*. This data may reside in another GPU's memory. It is up to the developer to initiate GPU-to-GPU copying or direct peer access.

Hedgehog is extensible; a specialized task can inherit from *AbstractTask* to create tasks, for example specialized for AMD GPUs using OpenCL or other libraries.

4.3 Nodes input and output data types

Nodes define their input and output types explicitly as template parameters (bound at compile-time) to ensure consistency. The input and output types of two nodes must match in order to connect them by an edge in the Hedgehog graph. Establishing this edge connection will automatically create the queues that are used to store data between the two nodes. A node is restricted to a single output type and can have multiple input types. The input and output types can be anything including types created by a user. For example, if a graph's node requires multiple output types, then these types can be gathered into a single class/struct. The input data types each have their own independent queue. The output of a task can connect to zero or more inputs of other tasks. Data that is sent out of a task is shared with all of its successor tasks. Hedgehog uses smart pointers to avoid the need for a deep copy. Data races that may result from multiple tasks receiving the same pointer are not controlled by the library. It is important for end-users to be aware of their usage patterns on received data.

For example, the accumulate (*Acc*) task in the matrix multiplication data-flow graph from Figure 1 contains a single input and two output edges. Instantiating this task using Hedgehog is done by specifying $Pair_{PC}$ as the input type and Blk_C as the output type for the node. An edge is then added from $State_{PC}$ to the *Acc*, and from *Acc* to both $State_C$ and $State_{PC}$. The edges will create queues for each of these nodes, if they were not already created, and any data that is produced from the *Acc* task will be shared with the $State_C$ and $State_{PC}$ tasks. In addition, the *canTerminate* function will need to be customized for the $State_{PC}$ or *Acc* nodes to break the cycle in the graph as will be discussed in the next section.

4.4 Threading model

A graph begins processing data when the graph is *executed*, which creates a thread for each task in the graph. If a task requests multiple threads (such as *MatMul* which accepts the number of threads at construction as shown in Listing 2), then custom copies of the task are made using the user-overloaded copy method. The original task and its copies form a task group, where each task

in the group is statically bound to a different thread. They share the same input and output edges consisting of queues and synchronization contexts. This forms contention on the input queues, but also offers higher throughput when sufficient data is waiting to be processed.

Hedgehog uses monitor synchronization to operate the edges; a consumer task holds a mutex and will enter the *wait* state if its input queue is empty. The monitor is implemented with a *std::condition_variable*. It will suspend the execution of the thread if no input data is available. The thread will wake up when notified by another node, this occurs when termination is requested or input data becomes available. When the thread is waiting for data, it will not consume CPU resources. If data is available, then the thread will remove it from its queue. A producer task sends *data* and signals to its successor consumer task, which transitions the successor consumer task from the *wait* state to the *running* state. The running and wait states of a consumer task are managed entirely by the operating system. After a context switch, the signaled consumer task will retrieve the *data* from its input queue and call its user-defined *execute* function on the received *data*. A consumer task can avoid the *wait* state and potentially the context switch if *data* were already available in its queue. A consumer task in Hedgehog infinitely fetches *data* from its input or waits until its *canTerminate* function returns true; by default this happens when the input connection is terminated and the queue is empty. When termination occurs, the terminated task will signal to all other tasks in the same group to wake-up and terminate properly.

Selecting the degree of parallelism for a task helps improve CPU utilization when executing the graph. For example, in matrix multiplication, the *MatMul* task can operate with the number of threads equal to the physical core count, whereas the accumulation task can operate with a fraction of the number of physical cores because it is less computationally complex. Specifying a good balance between the number of physical cores and the number of copies of a task can help prevent over-subscription of the CPU for compute intensive tasks, assuming there is enough data waiting to be processed in the task's input queue. In our experience, over-subscription rarely occurs due to the nature of the data-flow approach and how tasks will enter a running state only when data is present in a graph with dozens of independent tasks and edges. This behavior may vary depending on the nature of the algorithm. Furthermore, developers find the idea of tuning the number of threads understandable.

4.5 Memory management

Memory management is often needed when GPU computations or limitations due to hardware are involved. Hedgehog implements a memory manager, a tool used by tasks. An instance of a memory manager can be created with a specific data type and then attached to a task. This memory manager instance is thread-safe and limits the amount of data being produced by the task. If the task is copied, such as when building a task group, then all tasks in the group will share the same memory manager.

A memory manager creates a fixed-size pool of user-specified data buffers when the task is initialized. During a task's *execute* invocation, the task can fetch memory from this pool and will block if the pool is empty. The API provides a mechanism to recycle memory back to the memory manager, which will add memory back into the pool and signal to a task that is waiting. This allows a waiting task to obtain the needed memory and resume execution. The recycle mechanism provides two steps: (1) complete the allocation-deallocation cycle to eliminate memory leaks and (2) update the state of the memory. The state is updated when memory is returned to a memory manager. The state of memory indicates when a memory buffer is ready to be recycled. If the memory is not ready to be recycled, then it will not be added to the memory pool. This functionality is useful to indirectly express memory locality, such as to keep memory resident on GPUs to avoid unnecessary data transfers.

Two types of memory managers exist: static and dynamic. Both managers will fill the pool with objects during task initialization. The dynamic memory manager will call the default constructor for the data objects whereas the static one will call a specialized constructor.

4.6 Profiling

Hedgehog provides profiling through a printer class. It is used to represent the current state of the graph. To measure the overhead of profiling, we compare the performance of Hedgehog's first tutorial, the Hadamard product [20], with and without profiling, and execute it 1000 times on $16k \times 16k$ matrices and $2k \times 2k$ blocks, as shown in Figure 4.

To measure the impact of the profiling, we propose the following statistical analysis.

Let X_1 and X_2 be the execution times with and without profiling, respectively. Given the high number of experiments and the central limit theorem, the experimental averages, $\overline{X_1}$ and $\overline{X_2}$, follow a normal distribution. Now define the stochastic variable X as $\dfrac{\overline{X_1} - \overline{X_2}}{\sqrt{\dfrac{S_1^2}{N} + \dfrac{S_2^2}{N}}}$. X follows a normal distribution of expectation $\mu_1 - \mu_2$ and variance $\sigma^2 = 1$. Let S_1^2 and S_2^2 be estimators of σ_1^2 and σ_2^2.

For the null hypothesis to hold, we propose $\mu_1 = \mu_2$; there is on average no statistical difference between a computation with and without profiling. X becomes a standard normal distribution.

The size of our sample is N = 1000. For this sample, the values of $\overline{X_1}$ and $\overline{X_2}$ are respectively $\overline{x_1} = 1742.28\,\text{ms}$ and $\overline{x_2} = 1743.06\,\text{ms}$ and the estimations of the variances σ_1^2 and σ_2^2 are respectively $s_1^2 = 14.03\,\text{ms}$ and $s_2^2 = 12.49\,\text{ms}$. With this sample, the value of X is $x = -1.32$, whose p-value is 0.4066. These results allow us to accept the null hypothesis.

We can therefore conclude that the average execution runtimes, with or without profiling, do not differ significantly. We believe this is because we gather performance metrics at the node level, which is far less intensive than fine-grained profiling approaches, such as measuring all function invocations.

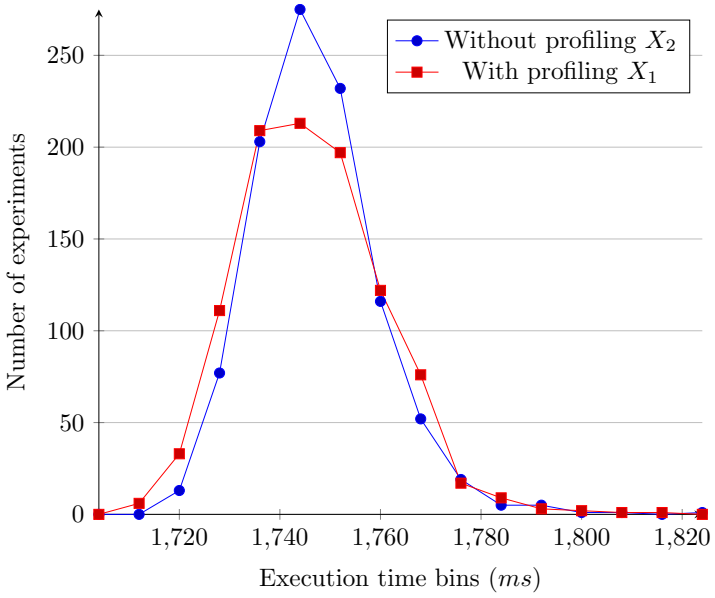

Figure 4: Hedgehog Tutorial 1 execution time distribution for 1000 experiments on $16k \times 16k$ matrices and $2k \times 2k$ blocks with and without profiling on a Mac Book Pro Mid 2015

The metrics gathered from the nodes are the execution time, waiting time, and queue usage. Additionally, the graph gathers the creation and total execution times. This information can be presented in various ways depending on options chosen by developers; for example, a task group can be expanded to show the performance of each thread. It is also possible to bind the graph to one or multiple POSIX signals with a *graph signal handler* to capture them and generate the state of the graph at that instant, such as when a segmentation fault occurs.

A DOT file printer has been developed to generate Graphviz [7] DOT files. The developer can use this visualization to understand how tasks and graphs interact with each other and using this representation can immediately recognize the critical path of the computation. Example DOT files can be found at the end of each of the Hedgehog tutorials [20].

4.7 Type checking

Hedgehog uses template metaprogramming to check the data-flow graph consistency at compile-time. It achieves static checking via constructs that use C++ 17 metafunctions [21]. This static checking will become simpler and more expressive with C++ 20 *constraints* and *concepts*.

4.7.1 Consistency of smart pointers

Hedgehog manipulates nodes by means of smart pointers based on an RAII (Resource Acquisition Is Initialization) technique, to avoid memory errors in the library. Basically, a smart pointer is an object wrapping a pointer of a given type. In C++, smart pointers are modeled by a generic class, `shared_ptr<T>`, where `T` is the type of the wrapped pointer.

To accept or reject a node, Hedgehog extracts the node's internal type from its smart pointer and checks this internal type's inheritance.

In Hedgehog, metafunctions have been designed to test the compatibility of two smart pointers.

4.7.2 Node compatibility

Hedgehog checks compatibility rules when two nodes are linked by an edge in the graph:

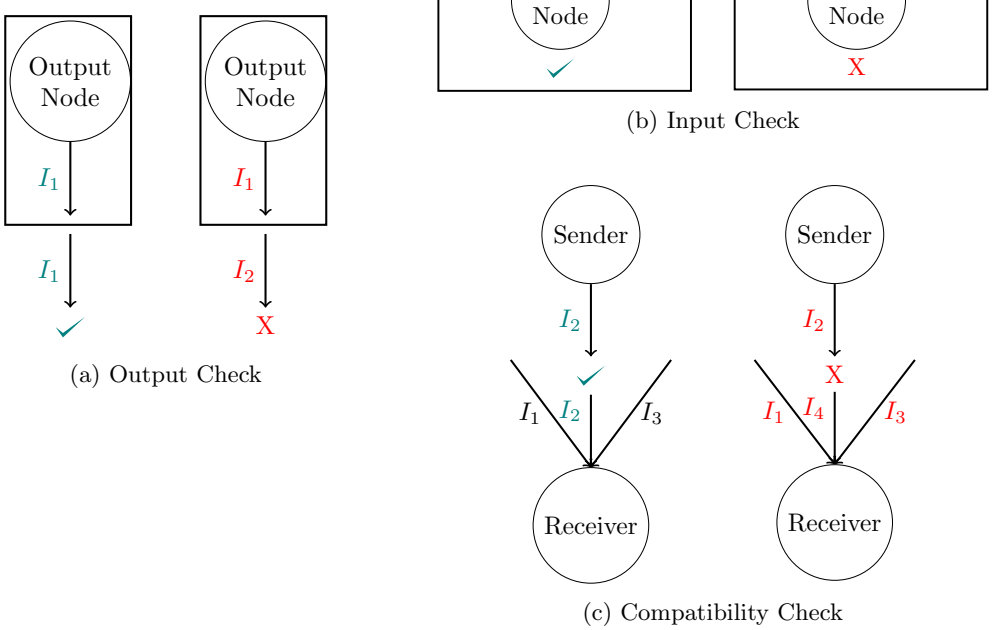

Figure 5: Compile-time check using metaprogramming techniques

1. If a node is set as the output of a graph, the node and the graph need to share the same output type, as shown in Figure 5a.
2. If a node is set as an input of a graph, the node and the graph need to share at least one input type, as shown in Figure 5b.
3. The output type of the source node must be one of the input types of the target node, as shown in Figure 5c.

Listing 3 shows the compatibility test (3), using a `static_assert` to perform the test and produce a clear error message at compile-time.

```
static_assert(
    traits::Contains_v<Output, Inputs>,
    "The given io cannot be linked together"
);
```

Listing 3: Metafunction invocation testing nodes compatibility

5 Results

The experiments conducted on Hedgehog help understand the behavior of the library, and the impact that having an explicit representation has on performance.

In our experiments, there are two crucial parameters, the amount of data streaming through the graph (often determined by decomposition parameters such as the block size in a matrix) and the number of threads for each Hedgehog node. The amount of data streaming through the graph affects the degree of parallelism that can be achieved. Because Hedgehog targets coarse-grain parallelism, it is important to determine an optimal decomposition size. For example, if the block size is "too small", then the underlying hardware may have poor utilization because each task does not execute enough instructions compared to the system latency. If it is "too big", then this will reduce the parallelism in Hedgehog because fewer pieces of data feed the tasks. The number of threads for each task will also determine how many elements a task can process in parallel. This can be detrimental if the processor gets oversubscribed. The best methodology for approaching computation in Hedgehog is setting good enough values for a first run. This run is used to generate the DOT file feedback, which identifies bottlenecks, and helps determine better parameters to improve the performance for a specific architecture. These graphs are portable, and only require parameter tuning for new architectures. This methodology justifies *experimentation for performance*, and was used to help identify the optimal parameters within our results.

5.1 Data transfer latency

One of the major costs in our approach is the data transfer latency between tasks. Two parameters affect this latency: (1) the number of threads per task, because all threads in a task group will share the same protected queues, which introduces access contention, and (2) the number of input types, because a task will have one queue per input type.

Figure 6 shows the results of measuring the time to transfer one element between two tasks, averaged over one million data transfers, on a computer with two Intel Xeon Silver 4114 CPU @ 2.20 GHz processors (10 physical cores, 20 logical cores) and 192 GiB of memory. This latency varies between $\approx 1\,\mu s$ to $10\,\mu s$. It grows with the number of threads in the group of receiving tasks and the number of input types. These measurements are low enough that the data transfer costs can be easily hidden by concurrently executing compute kernels. Furthermore, this latency is below that of *HTGS* for most common cases.

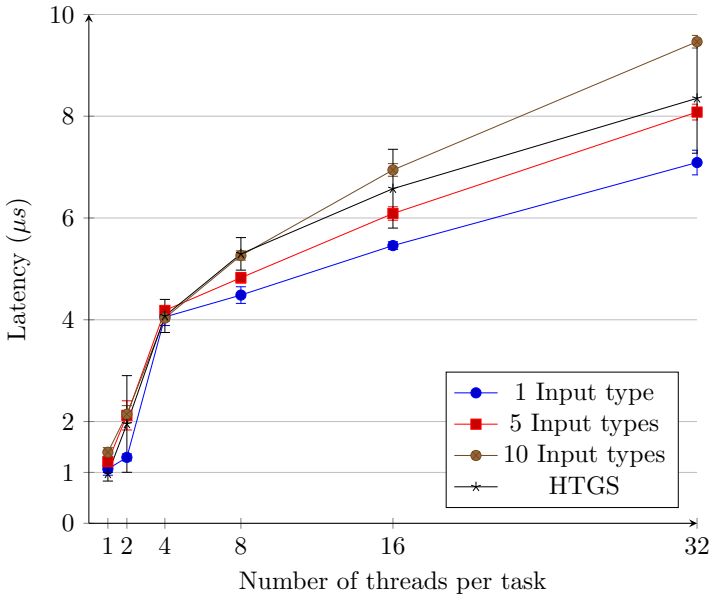

Figure 6: Latency analysis for different task's number of threads

5.2 LU decomposition with partial pivoting

One of our experiments was a CPU-based implementation of LU decomposition with partial pivoting [13].

The kernels used in our tasks are mainly composed of calls to the *OpenBLAS* [17] library in single-threaded mode. Our baseline was the *LAPACK dgetrf* routine, compiled with *OpenBLAS* in multi-threaded mode, used as a one-off call.

This methodology made it easier to reason about the complex dependencies and to instrument the code in a more sophisticated way that allows the streaming of data in and out of the operation. The stream-based design provides partial fully computed results before the computation completes. This is used to initiate the next step of a computation sooner.

To study end-to-end runtime performance, we ran 10 trials for the Hedgehog and *LAPACK* (Linear Algebra PACKage) implementations of LU decomposition with partial pivoting on two Intel Xeon E5-2680 v4 CPUs @2.40 GHz (28 physical cores, 56 logical cores) with 512 GiB of DDR4 memory. Figure 7 shows that Hedgehog obtains comparable overall performance to the baseline. In this case, the optimal parameter happens to be a block size of 1024. Additionally, the streaming aspect allows us to start getting partial results over 40 times faster than waiting for the entire matrix to finish computing. This has potential for future research in understanding the intricacies of chaining streaming operations together by composing graphs especially when it can be automated.

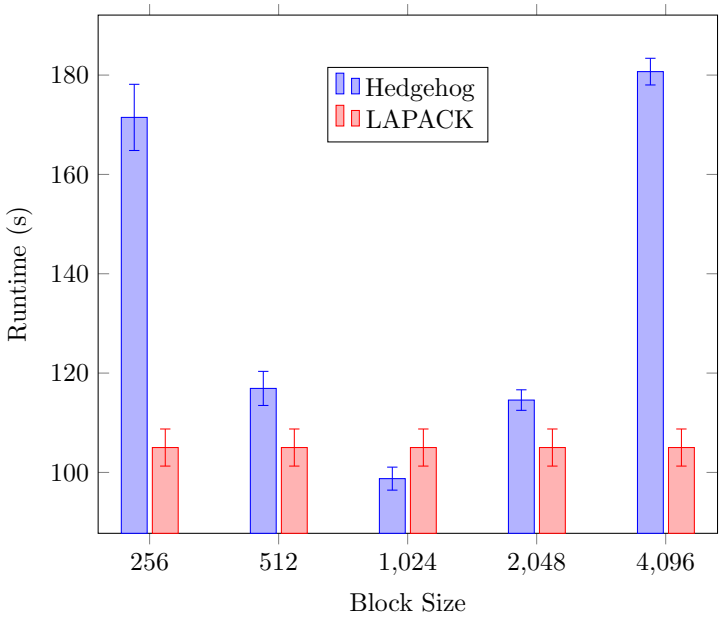

Figure 7: LU with partial pivoting performance for a matrix of 32768×32768 elements

5.3 Matrix multiplication

Hedgehog is a library that allows computation in a heterogeneous node. We developed a BLAS-like GEneral Matrix Multiplication routine (GEMM) and ran it on a node with two Intel Xeon Silver 4216 CPUs @2.1 GHz (16 physical cores, 32 logical cores) with 768 GiB DDR4 memory and four Tesla V100-PCIe with 32 GiB HBM2 GPU, using $128k \times 128k$ single-precision matrices. As shown in Figure 8, the Hedgehog implementation is compared to the *NVIDIA* counterparts from their GPU-accelerated libraries for basic linear algebra, *cuBLASMg* and *cuBLAS-XT* [5]. With an optimal block-size the Hedgehog implementation achieves $> 95\%$ of the theoretical peak across 4 GPUs.

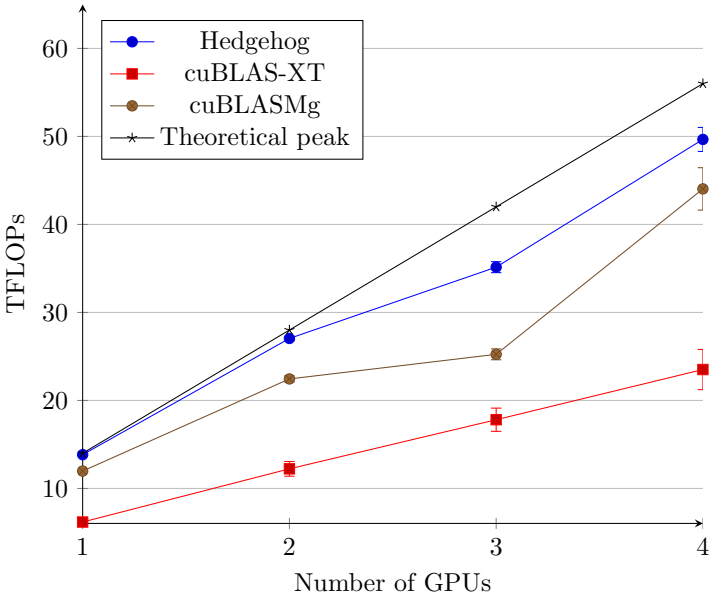

Figure 8: Matrix multiplication on a matrix of $64k \times 64k$ elements decomposed in $8k \times 8k$ block elements

This final result required multiple iterations that were done using Hedgehog's *experimentation for performance*. This methodology starts with a simple CPU-only version which is then augmented to use the GPU. The GPU version bundles the data transfers to and from the GPU and the actual computation into its own graph. This was made possible by the composability of the Hedgehog graphs. The GPU graph is further improved by inserting it into an *ExecutionPipeline* to scale the computation to multiple GPUs. These graphs could be represented as a library, which would simplify the future development of Hedgehog-based applications.

The CPU and GPU versions reused the original representation of state. The primary focus between the versions was optimizing the transfer of data between the CPU and GPU as well as improving the data locality to avoid unnecessary copies. For example, the *memory manager* was used to stay within the GPU memory limits, and *cudaEvents* were applied to memory copy events to overlap data motion, computation, and all of the state management that is done within Hedgehog.

These experiments show that the overhead incurred by our approach is not detrimental to performance. It is also amenable to using libraries that target fine-grained parallelism, such as calls to *cuBLAS* to maximize performance. Lastly, the explicit representation featured in Hedgehog is manageable. The LU decomposition with partial pivoting implementation was completed in the course of one month by a non-domain specialist without any prior knowledge of the Hedgehog library and the intricacies of parallel programming. This experience has continued to indicate to us that having a high-level programming abstraction that can be reasoned about and that maps to execution is extremely valuable.

6 Conclusion

In this paper, we have presented Hedgehog, a general-purpose library allowing developers to create a parallel computation on a heterogeneous node. It differs from other approaches as it only uses its explicit data-flow representation to implement an algorithm, and relies on data-pipelining to get performance without a global scheduler. This approach operates entirely based on the presence of data and achieves an overhead of $\approx 1\,\mu s$ to $10\,\mu s$ when transferring data between tasks in a graph.

Hedgehog supports a separation of concerns approach by providing several distinct components, such as computation, localized state, memory management, and scaling through execution pipelines. The graph is constructed using these components, and its representation is maintained throughout its execution. The developer can use this representation to understand the processing. This helps developers to reason about complex operations at a higher level of abstraction. This was demonstrated through the use of streaming data in and out of an operation to produce partial results as early as possible. This was achieved by explicitly representing the localized states of the computation throughout the algorithm, which aids with understanding when the results are finalized. In addition, Hedgehog can be easily extended as it relies on class inheritance to create new types of tasks, as we have presented for *CUDA*-based tasks.

Hedgehog operates effectively with heterogeneous computers as well, which is validated by our achieved GPU utilization. This is feasible by making use of Hedgehog's memory management tools, to express memory locality and avoid unnecessary memory copies and keeps GPUs busy.

7 Future work

Hedgehog has an explicit and static representation of the data-flow graph. The graph structure is known at compile-time. The latest C++ standard (C++ 20) brings more tools to perform complex computation at compile-time, such as additions to the *constexpr* specifier. A compile-time tool will be developed to analyze a Hedgehog graph structure to identify design mistakes such as data-races or cycles that cause deadlock. Another future work will be to extend Hedgehog to operate in cluster environments. One approach is to build one Hedgehog graph per node with direct interaction with external cluster-based communication tools and/or libraries.

Disclaimer

No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial software, products, and systems are identified in this report to facilitate better understanding. Such identification does not imply recommendations or endorsement by NIST, nor does it imply that the software and products identified are necessarily the best available for the purpose.

References

[1] Cédric Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurr. Comput. : Pract. Exper.* 23.2 (Feb. 2011), pp. 187–198. ISSN: 1532-0626. DOI: 10.1002/cpe.1631.

[2] Michael Bauer et al. "Legion: Expressing Locality and Independence with Logical Regions". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012, 66:1–66:11. ISBN: 978-1-4673-0804-5. DOI: 10.1109/SC.2012.71.

[3] Timothy Blattner et al. "A hybrid task graph scheduler for high performance image processing workflows". In: *Journal of signal processing systems* 89.3 (2017), pp. 457–467. DOI: 10.1007/s11265-017-1262-6.

[4] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. "Kokkos". In: *J. Parallel Distrib. Comput.* 74.12 (Dec. 2014), pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003.

[5] *Basic Linear Algebra on NVIDIA GPUs*. https://developer.nvidia.com/cublas. Last access: 2020-07-01.

[6] I. Culjak et al. "A brief introduction to OpenCV". In: *2012 Proceedings of the 35th International Convention MIPRO*. 2012, pp. 1725–1730.

[7] John Ellson et al. "Graphviz — open source graph drawing tools". In: *Lecture Notes in Computer Science*. Springer-Verlag, 2001, pp. 483–484. DOI: 10.1007/3-540-45848-4_57.

[8] M. Frigo and S. G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301.

[9] Alexandre Bardakoff et al. *Hedgehog Tutorials*. Sept. 2020. URL: https://pages.nist.gov/hedgehog-Tutorials/.

[10] J. A. Herdman et al. "Achieving Portability and Performance through OpenACC". In: *2014 First Workshop on Accelerator Programming using Directives*. 2014, pp. 19–26. DOI: 10.1109/WACCPD.2014.10.

[11] Hartmut Kaiser et al. *STEllAR-GROUP/hpx: HPX V1.4.1: The C++ Standards Library for Parallelism and Concurrency*. Version 1.4.1. Feb. 2020. DOI: 10.5281/zenodo.3675272.

[12] Laxmitant V. Kalé et al. *The CHARM Parallel Programming Language and System: Part I – Description of Language Features*. Parallel Programming Laboratory Technical Report 95-02. Last access: 2018-01-02. Parallel Programming Laboratory Department of Computer Science University of Illinois Urbana-Champaign, 1994. URL: http://charm.cs.uiuc.edu/papers/CharmSys1TPDS94.pdf.

[13] Gerson C. Kroiz et al. "Study of Exploiting Coarse-Grained Parallelism in Block-Oriented Numerical Linear Algebra Routines". In: *Proceedings of the 91st Annual Meeting of the International Association of Applied Mathematics and Mechanics*. Submitted to https://jahrestagung.gamm-ev.de/. 2020.

[14] Alexey Kukanov and Michael J. Voss. "The Foundations for Scalable Multicore Software in Intel Threading Building Blocks". In: *Intel Technology Journal* 11 (2007). ISSN: 1535-864X. DOI: 10.1535/itj.1104.05.

[15] Qingyu Meng and Martin Berzins. "Uintah Hybrid Task-Based Parallelism Algorithm". In: *Proceedings of SC12*. IEEE, 2012. DOI: 10.1109/SC.Companion.2012.237.

[16] *The OmpSs-2 Programming Model*. https://pm.bsc.es/ompss-2. Last access: 2018-08-16.

[17] Zhang Xianyi and Martin Kroeker. *OpenBLAS: An optimized BLAS library*. http://www.openblas.net/. Last access: 2020-05-22.

[18] *OpenMP API Specification: Version 5.0 November 2018*. Sept. 2020. URL: https://www.openmp.org/spec-html/5.0/openmp.html.

[19] Jonathan Ragan-Kelley et al. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176.

[20] *Tutorial 1 - Simple Hadamard Product*. https://pages.nist.gov/hedgehog-Tutorials/tutorials/tutorial1.html. Last access: 2020-07-10.

[21] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide (2nd Edition)*. 2nd. Addison-Wesley Professional, 2017. ISBN: 0321714121.

[22] Q. Wang et al. "AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs". In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12. DOI: 10.1145/2503210.2503219.