Hector: An Efficient Programming and Compilation Framework for Implementing Relational Graph Neural Networks in GPU Architectures

Kun Wu*
kunwu2@illinois.edu
University of Illinois at Urbana-Champaign
USA

Mert Hidayetoğlu
merth@stanford.edu
Stanford University
USA

Xiang Song
xiangsx@amazon.com
AWS AI
USA

Sitao Huang
sitaoh@uci.edu
University of California, Irvine
USA

Da Zheng
dazheng2@amazon.com
AWS AI
USA

Israt Nisa
nisaisrat@amazon.com
AWS AI
USA

Wen-mei Hwu
whwu@nvidia.com
Nvidia
University of Illinois at Urbana-Champaign
USA

Abstract

Relational graph neural networks (RGNNs) are graph neural networks with dedicated structures for modeling the different types of nodes and edges in heterogeneous graphs. While RGNNs have been increasingly adopted in many real-world applications due to their versatility and accuracy, they pose performance and system design challenges: inherent memory-intensive computation patterns, the gap between the programming interface and kernel APIs, and heavy programming effort required to optimize kernels caused by their coupling with data layout and heterogeneity. To systematically address these challenges, we propose Hector, a novel two-level intermediate representation and its code generator framework that (a) *captures* the key properties of RGNN models, and opportunities to reduce memory accesses in inter-operator scheduling and materialization, (b) *generates* code with flexible data access schemes to eliminate redundant data copies, and (c) *decouples* model semantics, data layout, and operators-specific optimizations from each other to reduce programming effort. By building on one general matrix multiply (GEMM) template and a node/edge traversal template, Hector achieves up to 9.9× speed-up in inference and 43.7× speed-up in training compared with the state-of-the-art public systems on select models, RGCN, RGAT and HGT, when running heterogeneous graphs provided by Deep Graph Library (DGL) and Open Graph Benchmark (OGB). In addition, Hector does not trigger any out-of-memory (OOM) exception in these tests. We also propose linear operator reordering and compact materialization to further accelerate the system by up to 3.8×. As an indicator of the reduction of programming effort, Hector takes in 51 lines of code expressing the three models and generates a total of 8K lines of CUDA and C++ code. Through profiling, we found that higher memory efficiency allows Hector to accommodate larger input and therefore attain higher throughput in forward propagation, while backward propagation is bound by latency introduced by atomic updates and outer products.

CCS Concepts: • **Computer systems organization** → **Parallel architectures**; • **Computing methodologies** → **Knowledge representation and reasoning**; • **Software and its engineering** → **Compilers**; **Massively parallel systems**; **Domain specific languages**.

ACM Reference Format:
Kun Wu, Mert Hidayetoğlu, Xiang Song, Sitao Huang, Da Zheng, Israt Nisa, and Wen-mei Hwu. 2024. Hector: An Efficient Programming and Compilation Framework for Implementing Relational

*Significant portion of the work is done during internship at AWS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0386-7/24/04
https://doi.org/10.1145/3620666.3651322

Graph Neural Networks in GPU Architectures. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620666.3651322

1 Introduction

Graph neural networks (GNNs) have been increasingly deployed in various applications, including fraud detection, recommendation, etc. In response to this growing demand, the open-source community has made much effort to provide GNN-specific machine learning frameworks, e.g., Deep Graph Library (DGL) [33] and PyTorch Geometric (PyG) [7]. These frameworks implement several highly-optimized operations, e.g., sparse-dense matrix multiply (SpMM) and sampled dense-dense matrix multiply (SDDMM), to speed up the execution [12]. Most of these operators and optimizations are for homogeneous graphs [12, 14, 38]. However, real-world graphs are typically heterogeneous by nature and contain multiple types of nodes and edges. For example, a citation graph may represent entities involving authors, articles, etc., as nodes of different types; the edges may model various types of relations, e.g., an article citing the others. Recently, to incorporate the information provided by such heterogeneity, relational GNNs (RGNNs) [13, 31] are proposed to define dedicated parameters and data paths for each type.

RGNN poses three major challenges to the existing GPU computation stack due to its inherent computation patterns, the gap between the programming interface and the kernel APIs, and the high cost of kernel code optimizations due to its coupling with data layout and heterogeneity.

The first challenge with GNN implementations on GPUs stems from their need to traverse graphs and scatter/gather tensor data in order to use high-performance general matrix multiply (GEMM) kernels to implement message passing. In RGNN, message passing is the procedure in each layer where an edgewise operation is followed by a nodewise aggregation operation. In other words, messages are passed through edges to the destination nodes. We show how message passing works in models in Section 2.1. During message passing, the graph structure and data layout significantly impact the memory access patterns and execution throughput [34, 39]. (Examples and details are in Section 3). Furthermore, as the connectivity of the input graph is involved in the gather computation, the computation patterns of GNNs are affected not only by the model definition but also by the graph. Such data-dependent behavior precludes any one-size-fits-all optimization strategy when executing GNNs. Additionally, RGNN introduces new complications into the design space due to the need for the operations to account for heterogeneity. We detail this in Section 2.

The second challenge in RGNN implementation stems from the lack of an abstraction layer between the programming interface and kernel APIs, resulting in extra data movement. A typical example is an edgewise typed linear layer. We detail the context and cause of the extra data movement in the edgewise typed linear layer in Section 2.3. But essentially, an edgewise typed linear layer multiplies one of the vectors on each edge with the layer weight dedicated to the edge type. To achieve this, many existing Pytorch-based systems materialize a temporary three-dimensional edge-wise weight tensor, where the slice corresponding to each edge is the weight matrix of its edge type. This temporary weight tensor is huge, causing redundant data access and memory footprint. Hector avoids such unnecessary copying activities by having typed linear transformations operate on views of tensors, a feature that PyTorch lacks, and decouples the materialization of its operands from the source-level expression (Section 3.2.2).

Third, code generation is necessary. High-performance neural network implementations have historically been based on pre-built libraries, e.g., cuBLAS [28]. GNNs make this less practical because the number of kernels to optimize is multiplied by the number of adjacency-matrix storage format choices such as Blocked-Ellpack [22]. For instance, cuSPARSE only supports the latter in a few APIs [27]. The typed edges and nodes of RGNN further exacerbate the problem, which makes the traditional pre-built libraries even less adequate and compels framework developers to either painstakingly develop optimized layers from scratch or settle for slow implementation. For example, it took more than a month for a full-time engineer to implement and deploy the typed linear layer of RGNN in DGL [24]. Another consequence is the performance degradation caused by limited kernels due to high implementation costs. For example, the DGL `HeteroConv` operator uses a Python native loop to separately launch kernels for each of the relation types in a heterogeneous graph, leading to serial execution of small GPU kernels that underutilize GPU resources on small graphs.

To systematically address these challenges, we propose Hector, a two-level intermediate representation (IR) and an associated code generator framework. The higher-level IR, called inter-operator level IR, defines the model semantics as sets of operators and expresses layout choices in a decoupled manner. At the lower level, the intra-operator level IR provides the facility to express template specialization and lower them to CUDA kernels. We further propose two optimizations, i.e., compact materialization (Section 3.2.2) and linear operator reordering (Section 3.2.3). We show in the corresponding Sections how these two optimizations are conveniently enabled by the two-level IR design. Sections 3.2 to 3.4 further the design and rationale of the two-level IR.

In short, Hector 1) represents the key properties of RGNN models to capture opportunities to reduce memory accesses

in inter-operator scheduling and materialization, 2) generates code flexibly with proper data access schemes to eliminate redundant data movement, and 3) expresses model semantics, data layout, and operator-specific optimization in a decoupled manner to reduce programming effort. To the best of our knowledge, Hector is the first to propose a multi-level IR to capture RGNN-specific opportunities involving cross-relation inter-operator optimizations and tensor data layout with consideration of the type dimension added by RGNNs. The contribution of this work is as follows.

1. We propose the Hector two-level IR and code generation framework to systematically optimize and generate GPU kernels for RGNN training and inference. It bridges the gap between the programming interface and the kernel generation process, decouples models, data layout, and operator-specific schedule from each other, and leverages optimization opportunities from the three aspects.
2. We devised the Hector code generator based on two generalized CUDA templates, i.e., a GEMM template and a node and/or edge traversal template. The generated code achieves up to 9.9× speed-up in inference and up to 43.7× speed-up in training compared to the best among the state-of-the-art systems [9, 35, 36] when running RGCN, RGAT, and HGT [2, 13, 31] on heterogeneous datasets provided by DGL and Open Graph Benchmark (OGB) packages [1, 4–6, 11, 32]. Hector also encountered fewer out-of-memory (OOM) errors, which is significantly alleviated by the optimization mentioned in Section 3. In fact, with compaction enabled, Hector incurs no OOM error for all the datasets tested in this paper.
3. We devised two optimizations: compact tensor materialization and linear operator reordering. The best combination of options varies across models and datasets and further obtains up to 3.8× speed-up in inference and 2.7× speed-up in training compared to our basic generated code mentioned in Contribution 2. Through profiling, we found that the improved memory efficiency allows Hector to accommodate larger computation and improve GPU hardware utilization for forward propagation. In contrast, backward propagation does not benefit from larger input, due to its latency-bound nature caused by atomic updates and outer products.

Artifacts are available at https://github.com/K-Wu/HET.

2 Background and Motivation

2.1 RGNN Formulation and Operators

GNNs are feed-forward neural networks that propagate and transform features using the connectivity of a graph. A widely used model is graph convolutional network (GCN) [16]. Formally, a GCN layer is defined as $\overrightarrow{h^{(l+1)}} = \sigma\left(A^* \overrightarrow{h^{(l)}} W^{(l)}\right)$, where $W^{(l)}$ denotes a trainable weight matrix of the l-th layer, σ is an activation function and $\overrightarrow{h^{(l)}}$ is the l-th layer node representation. A^* is the adjacency matrix normalized by node degrees. $\overrightarrow{h^{(0)}}$ denotes the node input features.

RGNNs extend GNNs to model different node and edge types for relational graph data. For example, extended from GCN, a relational graph convolutional network (RGCN) layer is defined as

$$\overrightarrow{h_v^{(l+1)}} = \sigma\left(\overrightarrow{h_v^{(l)}} W_0^{(l)} + \sum_{r \in R} \sum_{u \in \mathcal{N}_v^r} \frac{1}{c_{v,r}} \overrightarrow{h_u^{(l)}} W_r^{(l)}\right) \quad (1)$$

, where $\mathcal{N}_v^r$ denotes neighbors of node v in relation $r \in R$, $h_n^{(l)}$ is the l-th layer node representation of n. $W_r^{(l)}$ is the weight for relation r. $c_{v,r}$ is a problem-specific normalization factor. Figure 1 shows an example of how output features is produced in the message passing formulation equivalent to Formula 1: The forward propagation of an RGNN layer could be divided into ① the edge message generation stage and ② the node aggregation stage. For simplicity, we focus on the output feature $\overrightarrow{h_z^{(out)}}$ of node z: To obtain $\overrightarrow{h_z^{(out)}}$, ① a message $\overrightarrow{msg}$ is generated for each incoming edge, and ② the edge messages go through weighted aggregation and an activation function σ to produce $\overrightarrow{h_z^{(out)}}$. Notably, to obtain the output feature of node v, the input feature of v itself is applied to the $W_0^{(l)}$ and added to the transformed neighbor features. We call this a virtual self-loop because it could be seen as if each node now has a new edge to itself.

Relational graph attention network (RGAT) [2] and heterogeneous graph transformer (HGT) [13] are shown in Figure 2. Attention is introduced in these more complex models: Attention is produced in the message generation stage together with edge messages. Similar to the normalization factor, it is a scalar that emphasizes the message associated with the same edge during the subsequent node aggregation stage. However, attention is learned, as is produced by operations among weights and features.

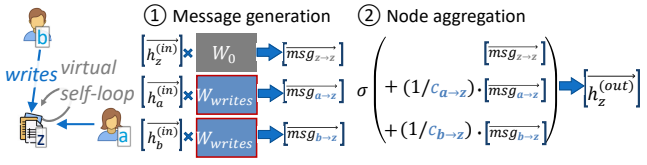


Figure 1. The forward propagation of an RGCN layer could be divided into ① message generation on edges and ② node aggregation. We focus on paper node z in a large citation graph as an example. z only has two incoming edges, from a and b, respectively. $\overrightarrow{h^{(in)}}$ and $\overrightarrow{h^{(out)}}$ are node features. W_{writes} is the weight for the type "writes". W_0 is the weight for virtual self-loops. σ is the activation function. Notably, some runtime implementations may replicate data, e.g., W_{writes}.

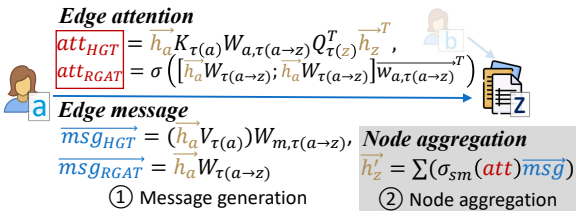


Figure 2. HGT and RGAT layer. $\overrightarrow{h_n}$ and $\overrightarrow{h'_n}$ are node n's features. Denote the type of edge from a to z as $\tau(a \rightarrow z)$. Weights $W_{a,\tau(a\rightarrow z)}$ differ by edge type $\tau(a \rightarrow z)$: For example, assuming there are two edge types, "writes" and "cites", $W_{a,\text{"writes"}}$ is a different weight from $W_{a,\text{"cites"}}$. They are defined and learnt according to the edge type. $W_{m,\tau(a\rightarrow z)}$ and $\overrightarrow{w_{a,\tau(a\rightarrow z)}}$ are in similar situations. Weights $W_{\tau(n)}$ differ by the node type $\tau(n)$ of n. σ is a leaky rectified linear unit (ReLU) in the case of RGAT. σ_{sm} stands for edge softmax. $[\vec{s};\vec{t}]$ concatenates $\vec{s}, \vec{t}$.

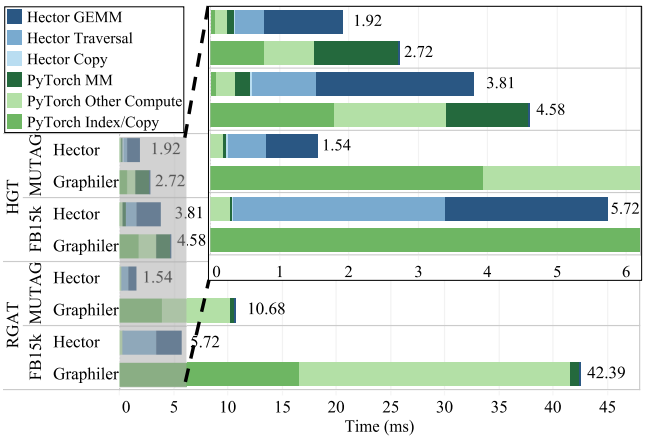


Figure 3. Breakdown of inference time by Graphiler and Hector. Matrix multiply (MM) includes SpMM. We categorize PyTorch time not accounted for by kernels as "PyTorch Other Compute".

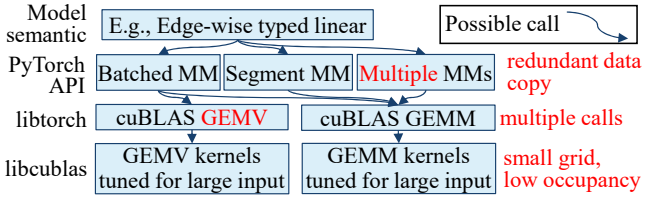


Figure 4. Inefficiency (in red) exists in all layers of existing systems.

2.2 RGNN Performance Characteristics

In non-graph neural networks, most linear operators, e.g., convolution, can be efficiently implemented with GEMM kernels. GEMM takes up most of the execution time due to its cubic complexity. While some operators can be optimized by transformations, e.g., Winograd for convolution layers [21], the operators are still computation-intensive after such computation reduction. GPUs are excellent at GEMM because the latter's high computation complexity allows leveraging the massive parallel compute units on GPUs, while the input data could be sufficiently reused to allow the memory bandwidth to keep up with the computation throughput.

In contrast, GNNs spend a much larger portion of their execution time on memory-intensive, non-GEMM operations [34, 39]. One major source of memory-intensiveness is the sparsity of graphs: to be not bound by the memory bandwidth, Nvidia H100 GPU requires the data reuse of single-precision float to be at least 16 times. However, the average degree of a graph often falls below this threshold, e.g., the graph datasets in Table 3. The heterogeneity of RGNNs further exacerbates the issue due to lowered data reuse by the introduction of dedicated weights to different edge types and node types, as shown in Figure 2.

2.3 Inefficiency in Existing Computation Stack: A Case Study on Edgewise Typed Linear Layers

We use an edgewise typed linear layer as an example to walk through the various performance overheads in the existing computation stack, as summarized in Figure 4. Edgewise typed linear layer applies a typed linear operator on each edge to one of its vectors. The weight of the linear operator used in the computation depends on each edge's type. For example, the edge message in an RGCN layer (Figure 1) or an RGAT layer (Figure 2), is produced by a typed linear layer.

A typed linear layer is typically implemented using batched matrix multiply (BMM) or segment matrix multiply (segment MM) [25]. For example, PyG `FastRGCNConv` implemented typed linear layers using BMM to unleash parallelism. However, a temporary tensor must be created from the weight tensor due to the lack of support for indirect addressing by PyTorch tensor APIs: the typed linear layer could be denoted as $Y[i,0,j] := \sum_k (X[i,0,k] \times W[T[i],k,j])$ where $X[i,0,\cdot]$, $Y[i,0,\cdot]$ and $W[T[i],\cdot,\cdot]$ are input feature, output feature of node i and the weight of node i's type. The middle dimension of X and Y are needed to make the operation a matrix multiply. However, there is currently no support for specifying $T[i]$ as one of the arguments to an operator in PyTorch; one must create $W'[i,k,j] := W[T[i],k,j]$ before the typed linear layer.

Segment MM requires presorting features by types. After that, the node/edge feature tensor is in the form of segments of features of the same type: the segment MM kernel then applies the corresponding weight tensor of the type to each segment. If neither BMM nor segment MM can be employed, one may fall back to multiple matrix multiplies, leading to higher device API overhead and GPU under-utilization.

Another type of inefficiency is suboptimal math library calls. PyTorch has routines to handle various scenarios, e.g., a tensor is strided in memory layout or is `NestedTensor`, a pack of tensors. Consequently, Pytorch sometimes performs BMM by launching multiple general matrix-vector multiplies

(GEMVs) kernels, which also leads to API overhead and GPU under-utilization.

Lastly, CUDA math libraries were initially developed for large inputs and may not be efficient for small inputs [28].

To better illustrate the points, Figure 3 breaks down HGT and RGAT inference time on FB15k and MUTAG. Section 4.1 details the system configurations and datasets. This experiment measured Graphiler [36], which executed compiled TorchScript code and delivered the best inference performance among the existing systems tested in this work. Figure 3 shows that indexing and copying take up a significant portion, and the portion of GEMM operations, i.e., MM vs. Other compute, varied with graphs. By profiling, we found that the CUDA API overhead is 22% the time of the critical path, which is the sum of the API overhead and kernel duration. This is partly due to a huge number of kernel launches caused by 1) libraries calling a series of kernels to fulfill an API invocation and 2) some operators calling separate sets of kernels for each types in the graph.

In contrast, Hector 1) **lowers more of the logic to GEMM**, and 2) assembles kernels with flexible access scheme to **gather and scatter data on the fly** to eliminate redundant data movement. Consequently, Hector does not replicate weights during computation. As shown, **this strategy achieves better performance than using hand-optimized kernels with dedicated functions to data movement, e.g., in Graphiler**.

To address the performance challenges in RGNN systems due to both RGNN's inherent computation pattern and the system design, we propose the Hector IR and code generation framework. By the IR design that *decouples* and *expresses* the model semantic, data layout, and operator-specific schedules, Hector opens up these opportunities and the integration of all three aspects into the design space. Table 1 shows the feature comparison of Hector with existing systems.

Table 1. Features of Hector and prior [9, 35, 36] GNN compilers.

		Graphiler	*Seastar*	*HGL*	***Hector***
Target	**Inference**	✓	✓		✓
	Training		✓	✓	✓
Memory efficiency		✓		✓	**better**
Design space	**Data layout**				✓
	Intra-operator schedule				✓
	Inter-operator optimization	✓	✓	✓	✓

3 Design and Implementation of Hector

3.1 Overview of Workflow and System Components

Hector consists of a programming interface, a code generator, and Python modules. The code generator takes in the model definition and generates both CUDA kernels and host functions that configure and invoke the CUDA kernels.

Figure 5 uses an example to illustrate the workflow. The input is an excerpt of DGL code invoking a typed linear layer on the input node features. Applying the `@hector.compile` decorator triggers a transpiling pass to lower the code into Hector inter-operator level IR. In this example, the typed linear transformation `typed_linear` can be efficiently implemented as GEMM kernels. To this end, Hector lowers the transform to an operator instance derived from the GEMM template at the inter-operator level. After the analysis and optimizations at the inter-operator level, Hector further lowers the code to a detailed GEMM specification at the intra-operator level. The GEMM output A collects edge data generated from the node data. The first input B is the weight matrix W, and the second input C is the collection of features of all the source nodes of the edges involved. The intra-operator level IR indicates that the GEMM operation should use the default tile width of 16 and be carried out without scatter, gather, or transpose applied to input or output matrices. Eventually, Hector generates a segment MM (Section 2.3) kernel, `gemm_1`. The Layout Choices section of Figure 5 shows the default layout choice. `etype_ptr` specifies the offsets of each segment of different type. `row_idx` is the source node index array in the COO format. The result tensor `e["msg"]` has the number of edges as the number of rows, and the number of the columns is the input dimension of the hidden layer. We detail in Section 3.2.2 an optimization technique, compact materialization, that is opened up by the decoupled layout choices from the inter-operator level IR.

The generated code is compiled into a shared library where host functions are exported through the `pybind11` utilities. Hector falls back to existing routines in PyTorch when certain operators are not yet supported. During runtime, the precompiled functions are loaded and registered as subclasses of PyTorch `autograd.Function`.

3.2 Inter-Operator Level IR

The inter-operator level IR follows the Python grammar but involves some new constructs, as listed in Table 2. Listing 1 illustrates how the attention calculation in a single-headed RGAT layer could be expressed using the inter-operator level IR. Lines 10-16 shows a code segment that generates attention values for all edges of graph `g` and then invoke the `edge_softmax(g)` function that spans lines 1 through 9. As shown in Listing 1, the message generation and aggregation stages are expressed as for-each edge loops starting from line 2, line 8 and line 10, and for-each node loop starting from line 4. To accumulate data from the incoming edges of each node n, the `n.incoming_edges()` iterator is used. Notably, the data layout that specifies how to access the input and output data per edge or node as well as the incoming edges associated with each node, is abstracted away in Listing 1.

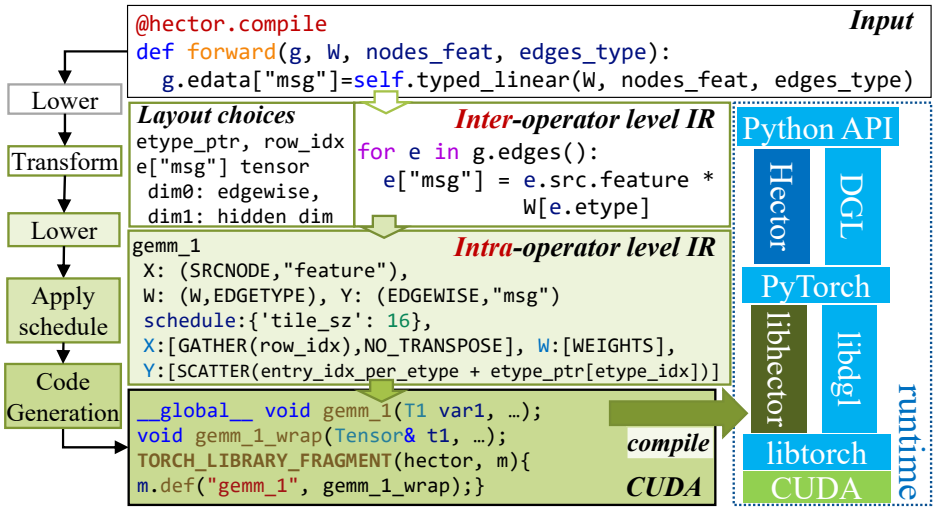
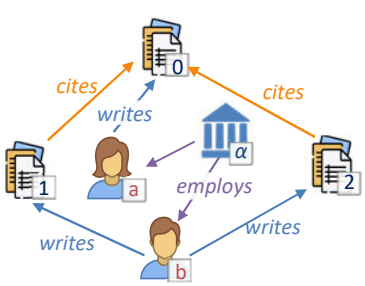


Figure 5. Hector workflow and software architecture.



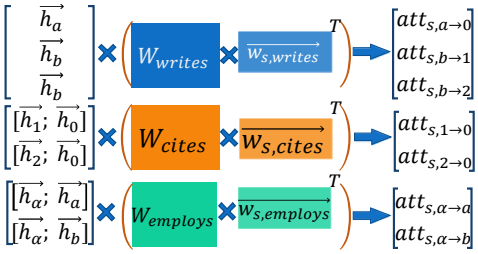
(a) The citation graph used in this example.

```
for e in g.edges():
  hs = e.src.feature * W[e.etype]
  atts = dot_prd(hs,w_s[e.etype])
  ht = e.dst.feature * W[e.etype]
  attt = dot_prd(ht,w_t[e.etype])
  e["att"] = leakyrelu(atts+attt)
```

(b) The original inter-operator level IR.

```
w_s_new = W[e.etype] *
    transpose(w_s[e.etype])
atts = dot_prd(hs, w_s_new)
```

(c) After the linear operator reorder, the gray region in (b) is rewritten.



(d) Visualization of computing `atts` in (b). Orange parentheses mark the computation order change after linear operator reorder.

Figure 6. In the example graph in (a), when computing edge attention of RGAT, linear operator reordering could be applied. (b) shows the original inter-operator level IR to compute RGAT edge attention. (d) visualizes the computation of the first term, `atts`, and uses the orange parentheses to mark how the linear operator reordering changes the order of the computation. (c) The transformation rewrites the code.

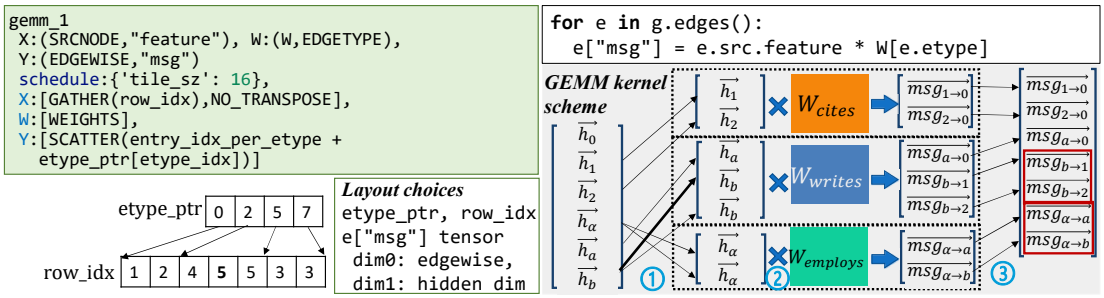
3.2.1 Programming Interface. Hector provides a decorator, `@hector.compile`, to take the existing PyG or DGL forward propagation logic and generate code for it, as exemplified by the input in Figure 5. The decorator, when applied to a method, invokes a simple transpiling pass that replaces the PyG and DGL method calls, e.g., SpMM/SDDMM, with an implementation in the inter-operator level IR, and replaces supported constructs from PyG and DGL with expressions in Hector IR. Similarly to statically-typed compilers in other Python packages [19, 29], the function to compile can use most of the Python features except dynamic ones, e.g., assigning objects of different types to the same variable. We support a few types as the function arguments for heterogeneous graphs, involving `Tensor` and `dict[str, Tensor]` objects, i.e., `dict` objects where the keys are `str` objects and the values are `Tensor` objects.

Besides, one can use the Hector inter-operator level IR itself to express the model, as exemplified by Listing 1.

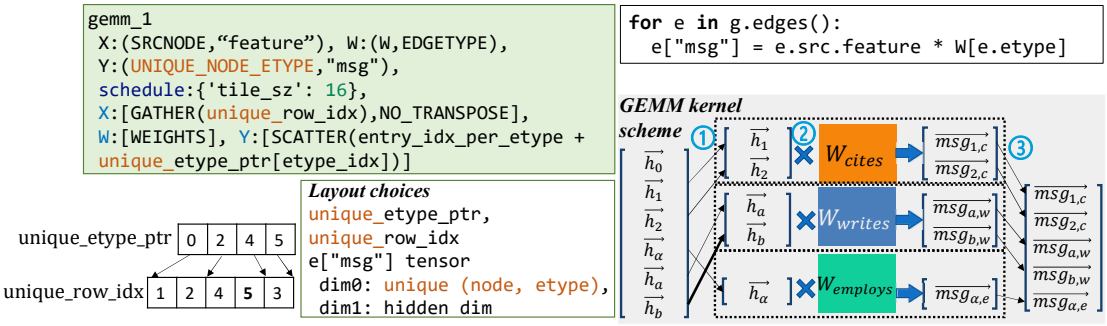
Listing 1. Expressing the attention calculation in a single-headed RGAT model using Hector inter-operator level IR.

```
def edge_softmax(g):
    for e in g.edges():
        e["att"] = exp(e["att"])
    for n in g.dst_nodes():
        n["att_sum"] = 0.0
        for e in n.incoming_edges():
            n["att_sum"] += e["att"]
    for e in g.edges():
        e["att"] /= e.dst["att_sum"]
for e in g.edges():
    hs = e.src.feature * W[e.etype]
    atts = dot_prd(hs, w_s[e.etype])
    ht = e.dst.feature * W[e.etype]
    attt = dot_prd(ht, w_t[e.etype])
    e["att"] = leakyrelu(atts + attt)
edge_softmax(g)
```

3.2.2 Compact Tensor Materialization and Data Layout. The Hector inter-operator level IR deliberately abstracts



(a) GEMM kernel and IRs of RGAT edge message computation with vanilla materialization. The two red squares mark identical terms because msg depends only on source node and edge type. Both schemes in (a) and (b) ① gather the source node's features into a matrix, ② perform the GEMM computation, and ③ scatter the output features to rows in the output tensor. Each dotted square mark a block in ② the GEMM kernel. row_idx specifies the source node index of each edge, and is used in step ①. etype_ptr specifies the offsets of edge of each type and is used in step ③.



(b) GEMM kernel and IRs of RGAT edge message computation with compact materialization. Differences in IRs are marked in orange.

Figure 7. When computing RGAT edge messages, compact materialization could be applied. This figure uses the graph in Figure 6(a). Compared with (a) vanilla materialization, (b) compact materialization saves both the memory footprint and the computation. In (a), the leading dimension of the output message tensor accommodates different edges. In (b), it accommodates unique (source node, edge type) pairs. unique_row_idx, and unique_etype_ptr describes the mapping from (source node index, edge type index) to the unique index.

away the data layout from the model semantics. As exemplified by Listing 1, the IR only expresses the association of variables with nodes or edges, e.g., e["att"] and n["att_sum"], without dictating the mapping of elements in the conceptual variable to the memory space.

In this work, we devised compact materialization, which is a technique enabled by the decoupling between model semantics and data layout. Note that certain edge data are determined by sparse combinations of source node features and edge types, e.g. $\overrightarrow{msg_{HGT}}$ in Figure 2. Rather than computing and storing such data for each edge, we instead compute and store the data once for each (edge type, unique node index) pair that actually exists, reducing the resources spent on computing and storing common subexpressions. As exemplified in Figure 7, the materialized tensor involves seven rows when each row vector corresponds to a msg of an edge. Alternatively, the system can materialize the tensor with only five rows, where each row vector corresponds to a msg of an (edge type, unique node index) pair. We call the former vanilla materialization and the latter compact materialization. For the vanilla scheme, the row number is the edge index specified by the sparse adjacency. For the compact scheme, it is a unique non-negative integer assigned to each (source node, edge type). We precompute this mapping and store it in a CSR-like format. Hector does not create the temporary weight tensor, as explained in Section 2.3. In summary, compact materialization is a technique to eliminate repetitive identical computations and results in edgewise operators. It is applicable when an edgewise operator depends only on the source node data and edge type, and its output has the shape of (number of edges, hidden dimension size). After this optimization, the output shape is reduced to (number of unique (*source node*, *edge type*) pairs, hidden dimension size), and repetitive computation is eliminated. Section 4.3 provided further analysis of the effects of compact materialization on memory footprint reduction.

Besides tensor materialization, the multi-level IR design also allows data layout optimizations involving 1) architecture-specific optimizations, e.g., padding, and 2) various sparse adjacency encoding. At the inter-operator level, data layout specifications are decoupled from the model semantics and do not influence the transform passes at this level. However, they determine the data access scheme and make a difference when generating CUDA code at the intra-operator level.

Table 2. Hector inter-operator level IR constructs. The graph's variable is named as g, node's as n, and edge's as e.

Methods of graph variables		
node iterator	`g.dst_nodes()`, `g.src_nodes()`	
edge iterator `g.edges()`	weight slicing, e.g., `W[e.etype]`	
neighbor iterator	`n.incoming_edges()`, `n.outgoing_edges()`	
Attributes		
nodes `e.src`, `e.dst`	types `e.etype`, `n.ntype`	
input data, e.g., `n.feature`	produced data, e.g., `e["att"]`	
Operators		
GEMM-eligible computation, e.g.,	`linear()`, `outer_prod()`	
GEMM-ineligible computation, e.g.,	`dot_prod()`	
manipulation, e.g.,	`reshape()`, `concat()`	

Hector inter-operator level IR bookkeeps the specifications, which are passed to the intra-operator level during lowering. The intra-operator level operator instances choose the data access scheme corresponding to the data layout specifications while assembling the kernel code. We leave the exploration of data layout optimizations to future work and detail our plan in Section 6.

3.2.3 Linear Operator Reordering. Linear operator reordering is an inter-operator level optimization. When a linear operator, e.g., linear layer and dot product, is followed by another linear operator, their order may be switched. For example, for `atts` as shown in Figure 6(d), we may calculate $W_r \vec{w}_r^T$ first instead. Its profitability can be determined by counting the number of multiplication and addition involved in the two GEMMs before and after the order is changed. For now, we implement the pass to switch the orders of two linear operators whenever this produces an operator between weights, because it reduces the complexity by reducing one of its factors, the number of nodes/edges, to the size of hidden dimension. For simplicity, rewritten operator instances use PyTorch BMM to compute the product of weights and apply PyTorch slicing when necessary.

3.2.4 Graph-Semantic-Aware Loop Transformation. Loop transformation at this level is augmented with the graph-semantic-specific equivalence rule: a for-each loop over the edges is equivalent to a for-each loop nest iterating over all the incoming/outgoing edges of all destination or source node. Loop transformation is applied during the lowering pass to canonicalize and fuse loops in order to more thoroughly identify kernel fusion opportunities.

3.2.5 Lowering Inter-Operator Level IR. To lower the IR to the intra-operator level, Hector greedily lowers every eligible operator to instances derived from GEMM templates (Section 3.3.1). Then, it fuses each remaining region and lower them to as few traversal instances (Section 3.3.1) as possible. To achieve this, Hector scans the code three times. Each time, it attempts to lower operators to instances of a specific preference level. During the first pass, it attempts to lower operators to GEMM-template-derived instances. In the next pass, it attempts the traversal-template-derived instances. The third pass will lower all the remaining operators to PyTorch function calls. During each pass, whenever an operator can be lowered, Hector marks the operator itself, together with all subsequent operators that can be fused into it, with the lowering decision. After all the operators have been examined in a pass, the marked operators are lowered and fused. Before the second pass, it canonicalizes the for loops and fuses loop nests whenever possible to discover kernel fusion opportunities.

3.3 Intra-Operator Level IR

The intra-operator level IR serves between the inter-operator level IR and the generated CUDA code. At this level, the IR should encode specifications to emit CUDA code and provide sufficient information specific to each operator invocation to the transform and lowering passes at the inter-operator level. The code transformation components at this level provide the methods to generate specialized CUDA code for the operators, to apply operator-specific schedules, and to return necessary information on operator selection and kernel fusion feasibility to the passes at the inter-operator level.

Hector's code generator ultimately lowers the IR to two basic constructs, the GEMM template and the traversal template. Algorithms 1 and 2 illustrate the edge traversal template and the GEMM template. The node traversal template is similar to Algorithm 2, and we will revisit it in Section 3.4.1. For simplicity, function template specialization refers to routines specialized for the specific instances derived from the two templates and involve 1) function arguments, e.g., number of rows, etc., 2) special registers, e.g., `threadIdx`, and 3) loop variables.

3.3.1 The GEMM Template and the Traversal Template. We base the code generation on GEMM and traversal templates because RGNNs involve not only sparse operations but also multiple dense operations to project vectors across different semantic spaces. The GEMM template serves edgewise and nodewise linear transformations, as exemplified by the computation of RGAT edge messages in Figure 7. The GEMM template is defined as a matrix multiply augmented with custom gather and scatter schemes. It is formulated as $Y[S] = X[G] \times W[T]$ where Y, X, W are output, input, and weights, respectively; S, G and T are scatter list, gather list, and the type of the nodes or edges, respectively. The traversal template performs generic nodewise or edgewise operations. It serves operators that cannot be lowered to GEMM templates, e.g., edgewise dot products.

As shown in Algorithm 1, the GEMM template is based on tiled matrix multiplication. The GEMM template starts with the work assignment per block during the `GetRange<kid>` subroutine (line 1). The `idxTileRow` and `idxTileCol` whose range is determined by `GetRange<kid>` is used to position

the workload. Typically, it is the coordinate of the tile of the output matrix. Factors that affect X's loading scheme, `LoadXToShmemIfInRange<kid>`, and W's, `LoadWToShmemOrRegistersIfInRange<kid>`, involve whether gather lists or transpose needs to be applied on the fly (lines 4-5). Gather list G in the Input section is sometimes needed to locate the rows in the source matrix X: For example, in Figure 7 (a), `row_idx` is needed in step ①. The required information will be passed during the lowering. The operator instance then accordingly chooses the data access scheme code piece for kernel code generation. The storing scheme `StoreCIfInRange<kid>` depends similarly on whether a scatter list will be applied. Atomic intrinsics are used in the case of multiple simultaneous updaters.

In the traversal template, as shown in Algorithm 2, the edge type, node indices retrieval scheme in lines 5-7 depend on the sparse adjacency encoding. Similarly to the GEMM template, when a row vector needs to be loaded or stored, the tensor materialization scheme determines how the row is located in the materialized tensor. All statements are initially inserted into the innermost loop. After Hector finishes the loop transformations, it then defines work assignment on line 1 in Algorithm 2 for the operator instance derived from the traversal template using a simple scheme. For example, if the loop nest is three levels, as exemplified by Algorithm 2, we assign the outermost loop, i.e., `idxEdge` or `idxNode` loop, to each thread block and the two inner loops to the multi-dimensional threads in each block.

Algorithm 1: Hector's GEMM template in pseudo-code. Each instance is assigned a unique identifier `kid` and gets function template specialization **`FuncName<kid>`**.

```
   Input: References of Tensor Y, X, W, gather list G, etc.
1  tileRowRange, tileColRange ← GetRange<kid>();
2  foreach idxTileRow ∈ tileRowRange do
3      foreach idxTileCol ∈ tileColRange do
4          LoadXToShmemIfInRange<kid>();
5          LoadWToShmemOrRegistersIfInRange<kid>();
6          __syncthreads();
7          Y_reg ← X_shmem × W_shmem_or_reg;
8          __syncthreads();
9      StoreYIfInRange<kid>();
```

3.3.2 Adapting to Different Sparse Adjacency Encoding.

At the intra-operator level, the templates work for any sparse adjacency encoding as long as specific interfaces are implemented. For example, the edge traversal shown in Algorithm 2 works as long as the function template specialization `GetEType<kid>`, `GetSrcId<kid>` and `GetDstId<kid>` are implemented: If the sparse adjacency is COO, `GetSrcId<kid>` is a subscript operator applied to the row indices array. If it is CSR, then `GetSrcId<kid>` is a binary search in the row pointer array.

Algorithm 2: Hector's edge traversal template in pseudo-code. Similarly to Algorithm 1, each instance gets specialized **`FuncName<kid>`**.

```
   Input: References of input and output tensors. Other
          necessary data, e.g., adjacency.
1  eRange, hRange, fRange ← GetRange<kid>();
2  foreach idxEdge ∈ eRange do
3      foreach idxHead ∈ hRange do
4          foreach idxFeat ∈ fRange do
5              eType ← GetEType<kid>();
6              srcIdx ← GetSrcId<kid>();
7              dstIdx ← GetDstId<kid>();
               // initial insertion point
```

3.4 Rationale of the Hector Two-Level IR

Central to the code generator is the two-level IR. Inter-operator level IR optimizations address the opportunities brought in by heterogeneous relation types. These optimizations manipulate operators and their connections. A high-level IR abstracts away the low-level details that can complicate or even hinder the transformations. Intra-operator level IR optimizations reduce the data movement by generating access schemes in kernels rather than using specialized kernels and dedicated indexing/copying kernels. These optimizations manipulate low-level data access and schedule details, and thus are better supported by a low-level IR.

The two-level IR enables concerted but decoupled choices of intermediate data layout and compute schedules: For example, in Figure 5, the semantics of the model are decoupled from the layout choices. Hector implements the model semantics and layout choices in intra-operator level IR with specific access schemes. The next few paragraphs explain how the two-level IR design facilitates operator-specific optimizations, operator selection, and kernel fusion.

3.4.1 Operator-Specific Schedule.

Each instance derived from the GEMM template provides the option to apply a coarsening factor in $\{2, 4\}$, to choose the tile size, and to apply `__launch_bounds__` that limits the number of registers in exchange for more active warps. The coarsening factor is the number of elements each thread deals with in the loading, computing, and storing stages. When applied, each block still works on the same assignment, but its number of threads shrinks by the factor [22]. We also allow a per-row scalar to be applied to the tiles of matrix A. This eliminates the extra memory-intensive traversal to perform weighted vector summation by attention or norm.

As for the traversal template, similarly to the discussion in Section 3.2.4, we incorporate graph-semantic-aware loop transformation rules that allow Hector to leverage graph semantics to open up the trade-off between more data reuse opportunities and greater parallelism. As mentioned in Section 3.3.1, initially, all statements are in the innermost loop

in each instance derived from the traversal template. Loop hoisting is performed to enhance data reuse: The template features insertion points before and after the end of each loop level. For each statement, Hector finds the outermost level where it can be placed before applying the template. In addition, the template also provides a partial result aggregation method, which is applied during lowering by default, to reduce global memory traffic by accumulating results within a thread and within a warp before atomically adding them to the data in global memory.

3.4.2 Operator Selection and Kernel Fusion. Transformation and lowering passes at the inter-operator level need information about operator instances, specifically operator preference and the feasibility of kernel fusion. Preference level is the mechanism Hector uses to select the operator instance when there are multiple candidates. For example, an operator instance derived from the GEMM template may have an alternative derived from the traversal template but the alternative would lead to lower performance due to much lower data reuse. For good performance, operator instances derived from the GEMM template are assigned a higher preference level than those derived from the traversal template unless otherwise specified. Instances that fall back to PyTorch have the lowest preference level.

Operator instances also provide methods to determine the feasible operators to be fused within the IR. Operator instances derived from the GEMM template can be fused with the consumer if 1) the latter multiplies the row vectors in the GEMM output with scalars and 2) the two operators are in the same loop (nest). Operator instances derived from the traversal template can be fused with each other as long as they are in the same loop (nest). If the inter-operator level pass finds that some temporary variables are created and merely used inside the fused operator, it passes that knowledge to the method so that the variable no longer needs to be created in the global memory.

3.5 Backward Propagation

Similarly to PyTorch, Hector supports auto-differentiation by maintaining the backward propagation counterparts of the operators. Hector first emits the backward propagation via inter-operator level IR, and removes unused gradients and their computation. The lowering and code generation schemes are similar to those in forward propagation. However, additional processing is needed because the PyTorch auto-differentiation requires the backward propagation methods to be paired with the forward propagation methods in the `autograd.Function` definitions. To achieve this, Hector bookkeeps the kernel calls in each forward propagation method. For each forward propagation method, Hector puts all the corresponding backward propagation kernel calls in the body of the backward propagation method.

3.6 Code Generation

The code generation procedure emits code based on the CUDA kernel specifications detailed in the form of intra-operator IR. Kernel code generation is fairly straightforward and is implemented using a template-based approach. Hector then emits the host functions that configure grids and blocks, gets raw pointers from the `libtorch` `at::Tensor` references, and launches the corresponding kernel. The host functions are exported via `pybind11` utilities.

The Hector performs a pass that scans all the functions generated to collect a list of preprocessing required for the input dataset, involving transposition, converting COO to CSR, etc. The code generator then emits the preprocessing code.

3.7 Applicability of the Optimizations to GNNs.

Linear operator reordering and compact materialization are specific to RGNNs. Linear operator reordering is specific to RGNNs because RGNNs typically require linear projections from different semantic spaces, introduced by the heterogeneity of node types and edge types, to a common space before further operations. Compact materialization is specific to RGNNs because of the additional tensor dimension brought in by different node types and edge types.

Some of the intra-operator IR optimizations could benefit ordinary GNNs, which can be treated as a special case of RGNNs whose relation type number is one. Intra-operator level IR allows specification of both data access schemes and schedules, thus allowing flexible code generation to accommodate different dense or sparse tensor layouts, a need that often arises from compact materialization. However, the ability to generate code for different data access schemes and schedules can be beneficial when compiling ordinary GNNs.

4 Evaluation and Discussion

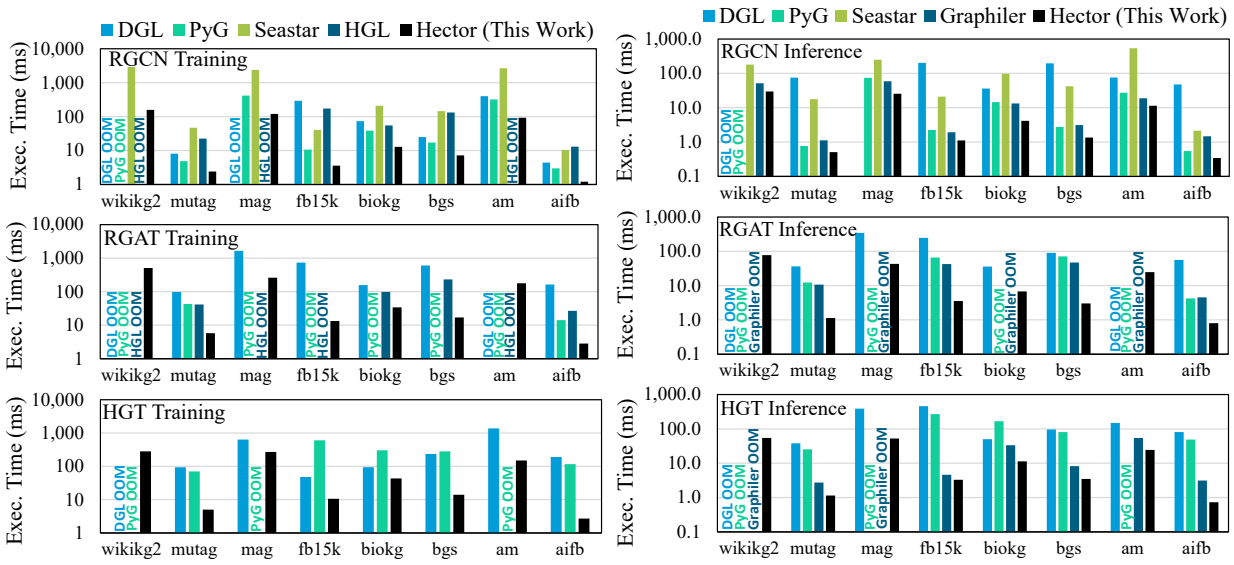
We evaluate Hector with the following questions to answer.

Q1. How does the performance of Hector compare with state-of-the-art systems? How does Hector achieve it?
Q2. How much improvement do the two optimizations detailed in Sections 3.2.2 and 3.2.3, compaction materialization and linear operator reordering, make?
Q3. Any architectural insights for GPU for RGNNs?

Section 4.2 answers Q1. Section 4.3 answers Q2, and further analyzes the performance implication of the two optimizations through a case study. Section 4.4 addresses Q3.

4.1 Methodology

To assess performance, we measure the inference and training time of Hector and other systems on a single-GPU computer. Its hardware components include one Intel Core i9-9900K CPU, 128 GB dual-channel memory, and one Nvidia RTX 3090 GPU with 24GB memory. The operating system is Ubuntu 18.04.5, with kernel version 5.4.0-135. The CUDA and



(a) Training time **(b)** Inference time

Figure 8. Comparing the execution (Exec.) time of Hector best optimized code with previous work. Table 3 shows the datasets used.

Table 3. Heterogeneous graph datasets [1, 4–6, 11, 32] used in our evaluation. The numbers reflect the default preprocessing by the OGB and DGL packages, e.g., adding inverse edges.

Name	# nodes (# types)	# edges (# types)	Name	# nodes (# types)	# edges (# types)
aifb	7.3K (7)	49K (104)	fb15k	15K (1)	620K (474)
am	1.9M (7)	5.7M (108)	mag	1.9M (4)	21M (4)
bgs	95K (27)	673K (122)	mutag	27K (5)	148K (50)
biokg	94K (5)	4.8M (51)	wikikg2	2.5M (1)	16M (535)

driver versions are 12.1 and 530.30.02, respectively. PyTorch and DGL versions are 2.0.1 and 1.1.1, respectively.

As shown in Table 3, we use public datasets from DGL [33] and OGB [11]. We measure (1) inference and (2) training time on three RGNN models, RGCN [31], RGAT [2], and HGT [13], comparing with previous systems, involving DGL [33], PyG [7], Seastar [35], Graphiler [36], and HGL [9]. We ported Seastar artifacts to the same version of CUDA and Python packages as what Hector depends on because one of Seastar's dependencies, dgl 0.4, used an API deprecated since CUDA 11.

For RGCN, RGAT, and HGT, excluding comments, Hector took in 51 lines in total and produced more than 3K lines of CUDA kernel code, 5K lines of other C++ code to define host functions, and 2K lines of Python code to define subclasses of Pytorch `autograd.Function`. The implementation also involves 2K lines of Python code providing common utilities.

To best align with the hyper-parameters prior work used in its evaluation, we set the input and output feature dimensions as 64 and the number of heads as 1. We measure the inference and training time of the single layer used. In training, to obtain a loss, we compute the negative log-likelihood loss by comparing the output with a precomputed random label tensor. For each case, we run the full graph inference and training for at least 10 epochs and average the elapsed time. To align with the existing system, nodes are presorted to enable segment MM for typed linear layers.

4.2 Comparison with Prior Work

For the performance of DGL and PyG, we measure all public implementations of these models from DGL, PyG, and Graphiler artifacts. PyG provides two RGCN convolution layers: `RGCNConv` places nodes in segments of the same type but launches separate kernels for each of the node types, leading to device underutilization. `FastRGCNConv` replicates weights and uses `bmm()`. It is consistently faster than the `RGCNConv` implementation. Similarly, DGL's built-in segmentMM-based RGCN layer is faster than other DGL implementations. For HGT, the DGL segmentMM-based `HGTConv` primitive generally has the best performance. In the cases where some variants encounter OOM errors, we choose the best among those that run without issues. Some cases are missing due to insufficient operator support, such as HGL on HGT and Graphiler on training. We do not measure HGL in inference because it is designed to optimize training.

Figure 8 shows that Hector's best-optimized code consistently outperforms state-of-the-art systems. It achieves up to 9.9× speed-up in inference and up to 43.7× speed-up in training against the best of state-of-the-art systems. On geometric average, Hector gets 1.79×, 8.56×, 2.87× speed-up in inference via RGCN, RGAT, and HGT, respectively, and 2.59×, 11.34×, 8.02× speed-up in training RGCN, RGAT, and HGT, respectively. The performance advantage is larger

in small graphs, demonstrating that **generating a single kernel that performs the computation across multiple edge types boosts the performance on small graphs compared to existing systems that run many small kernels**.

We see close performance achieved by Graphiler in RGCN and HGT inference. Graphiler leverages PyTorch utilities to produce TorchScript binaries before execution and utilizes edgewise parallelism for edgewise computation. Similarly to `RGCNConv`, it places node features into segments of the same type but runs separate kernels to perform a typed linear transformation. DGL and PyG under similar configurations achieve competitive performance. However, when it comes to RGAT, Graphiler suffers from performance degradation. Because Graphiler relies on pre-programmed fused kernels to deliver a significant portion of the performance boost [36], we postulate that the degradation is due to the non-exhaustiveness of these pre-programmed kernels [37]. This reflects the drawbacks of compiler design without a code generation mechanism. By contrast, with two-level IR and a code generator, Hector achieves better performance, showing that **generating kernels with flexible access scheme that gather and scatter data on the fly eliminates redundant data movement and outperforms indexing/copying followed by hand-optimized GEMM and sparse kernels**. Besides, it is challenging to extend Graphiler's approach to training due to TorchScript's limited auto-differentiation support. For example, `dict` object creation is not supported, but it is a common way to express nodewise data and edgewise data.

By comparing Hector with Seastar, which lowers all logic to sparse kernels, we realize that **sparse kernel code generation alone is not efficient in RGNNs: it is better to lower to GEMM kernels as much as possible**.

There are two reasons why Hector is more efficient in device memory usage. First, Hector only keeps a single copy of weights, as discussed in Section 3.2.2. Replicating weights also affects backward propagation because the gradient of each individual copy will be derived, occupying extra memory. Second, our compact materialization reduces memory and redundant computation, as explained in Section 4.3.

Notably, even without compact materialization or linear operator reordering, Hector still consistently outperforms existing systems, as Table 4 shows. In addition, the unoptimized Hector code triggers fewer OOMs than existing systems, with the only exception where the RGAT inference is run on mag and wikikg2. For comparison, we also show the statistics of the best optimized Hector code in Table 4.

4.3 Effects of Compact Materialization and Linear Operator Reordering

Now, we study the effects of compact materialization and linear operator reordering. They are detailed in Sections 3.2.2 and 3.2.3. We investigate their effects on RGAT and HGT.

Table 4. Comparing to the best in state-of-the-art systems, speedups of Hector unoptimized (unopt.) code and that of Hector best optimized (b. opt.) code. Worst (W), average (M), and best (B) cases. Numbers of OOMs Hector triggers (#E) are shown.

		Training				Inference			
		W	M	B	#E	W	M	B	#E
unopt.	RGCN	2.02	2.59	3.47	0	1.51	1.79	2.19	0
	RGAT	1.72	9.14	43.7	2	1.41	5.02	9.89	2
	HGT	1.53	6.62	28.3	0	1.20	1.90	4.31	0
b. opt.	RGCN	2.02	2.76	3.48	0	1.51	1.91	3.20	0
	RGAT	4.61	11.3	55.4	0	5.29	8.56	15.5	0
	HGT	2.17	8.02	43.1	0	1.40	2.87	7.42	0

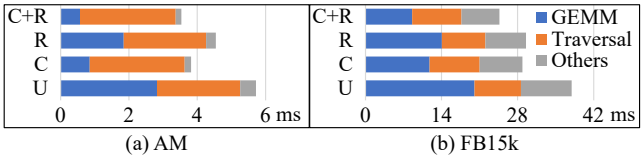


Figure 9. Breakdown of Hector RGAT inference on two datasets. Input and output dimensions are 64. Cases with compaction (C), linear operator reordering (R), and no optimization (U) are presented.

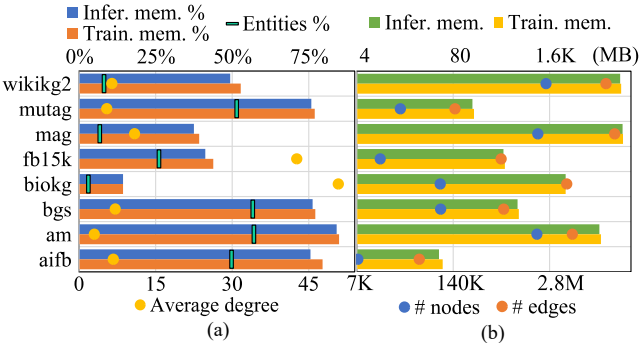


Figure 10. Memory usage when Hector runs training and inference on HGT. (b) shows the inference memory use (Infer. mem.) and training memory use (Train. mem.) of the unoptimized Hector code in MBs. (a) shows the portion of the memory use after applying compact materialization vs. the unoptimized Hector code. For comparison, the number of nodes (# nodes), number of edges (# edges), and average degree of datasets are shown as dot scatters. The entity compaction ratio of each dataset is also shown. Legend entries of each data series are placed next to the axis the series uses.

Table 5 shows the speed-up on top of Hector unoptimized code by these two optimizations. Due to compact materialization, when running RGAT on mag and wikikg2, Hector no longer triggers OOM errors. In addition, in some cases, the layout speeds up the execution as well due to the common subexpression elimination brought forth by the layout. Compact materialization is hardly possible without a code

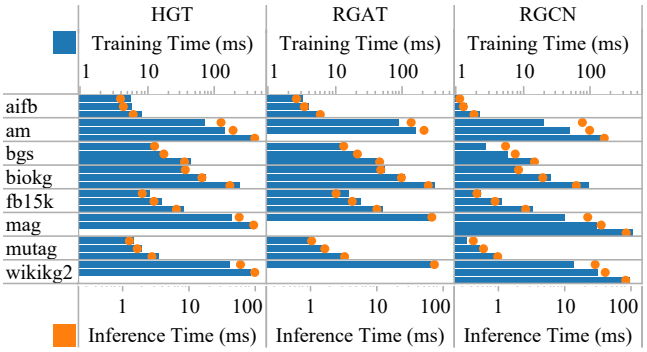


Figure 11. Hector unoptimized performance. Each cell corresponds to one pair of dataset and model, where it is shown the time of (input dimension, output dimension) as (32, 32), (64, 64), and (128, 128) from the top to the bottom. Vacancy indicates OOM errors.

generation scheme or an IR design that decouples the model semantics, data layout, and operator-specific schedule. Besides, **data layout choice, compact materialization in particular, allows further performance enhancement** while prior work usually focuses on the improvement of schedule given a specific sparse matrix format. This is shown by the great speedups in the "C[ompact]" columns in Table 5.

To study how compact materialization reduces the memory footprint, We illustrate the Hector dram usage without compact materialization in Figure 10(b) and the portion of dram usage with compact materialization in Figure 10(a). For simplicity, we define the entity compaction ratio as the number of unique (source node, edge type) pairs divided by the number of edges. Figure 10(b) shows that the memory use of inference and training is highly proportional to the number of edges of the datasets. Figure 10(a) shows that compact materialization significantly reduces DRAM usage in all datasets. The memory footprint ratio of compact materialization compared with the memory footprint of the unoptimized code correlates with the entity compaction ratio. The memory footprint ratio is higher than the entity compaction ratio, as the memory footprint consists of edgewise data, nodewise data, and weights, whereas the compaction applies to edgewise data only. Besides, in case the average degrees are larger, the memory footprint ratio reduces more significantly, getting closer to the entity compaction ratio.

To better understand the performance benefits of optimizations, Figure 9 studies two cases. The entity compaction ratio of AM and FB15k are 57% and 26%, respectively. On AM, the time GEMM instances take is greatly reduced. By comparison, in FB15k, compaction brings less performance improvement due to the less significant GEMM reduction.

In short, **due to the data-dependent nature of computation in RGNNs, there is no one-size-fits-all optimization strategy**. However, as shown in Table 5, Enabling compaction and reordering obtains fairly good performance consistently and is the best fixed strategy on average in all four scenarios, i.e., {RGAT, HGT} × {training, inference}. If Hector presumably chooses the best configuration in every run, it could further get 1.06×, 1.33×, 1.02×, and 1.08× speed-up in the four scenarios above, respectively. We leave autotuning to future work.

Table 5. Speed-up on top of Hector unoptimized code due to compaction (C) and linear operator reordering (R). Input and output dimensions are both 64. The highest speed-ups per task are in bold.

		Training			Inference		
		C	R	C+R	C	R	C+R
RGAT	aifb	0.80	**1.14**	0.84	1.01	**1.19**	1.10
	am	0.94	**1.12**	0.99	1.31	1.28	**1.54**
	bgs	0.93	**1.18**	1.04	1.29	1.34	**1.57**
	biokg	2.67	1.26	**2.68**	**3.76**	1.40	3.74
	fb15k	1.20	1.20	**1.27**	1.50	1.26	**1.62**
	mag	1.51	OOM	**1.57**	1.00*	OOM	**1.07**
	mutag	0.70	**1.14**	0.73	1.23	1.24	**1.36**
	wikikg2	1.09	OOM	**1.12**	1.00*	OOM	**1.02**
	AVERAGE	1.13	1.17	**1.18**	1.36	1.28	**1.49**
HGT	aifb	0.97	**1.52**	1.40	0.92	**1.94**	1.58
	am	1.05	1.12	**1.19**	1.06	1.32	**1.42**
	bgs	1.00*	1.11	**1.18**	0.94	**1.25**	1.24
	biokg	1.35	1.03	**1.41**	1.45	1.07	**1.58**
	fb15k	0.88	**1.11**	0.96	0.77	**1.16**	0.86
	mag	1.24	1.06	**1.34**	1.46	1.10	**1.72**
	mutag	1.00	**1.32**	1.32	0.94	**1.68**	1.50
	wikikg2	1.22	1.07	**1.33**	1.26	1.15	**1.51**
	AVERAGE	1.08	1.16	**1.26**	1.07	1.31	**1.40**

*Normalized by the performance with compact materialization (C) because the unoptimized version triggers OOM errors.

4.4 Analyzing the Architectural Characteristics

We show the average time of unoptimized Hector in Figure 11. We also further profile generated kernels when running Hector on RGAT on bgs and am, as shown in Figure 12.

One thing to note is the sublinear time increase in Figure 11: when the input and output dimension doubles, the amount of computation and memory accesses becomes close to 4× those of the original, but the time increase is typically lower than 2× of the original. The reason is increased computation throughput when the size increases, as corroborated by Figure 12. Moreover, we observed higher throughput when the graph scale increases, e.g., from bgs to am in Figure 12. Similarly, we witnessed the cuBLAS throughput increases steadily when we keep the right matrix size as (64, 64) and increase the number of rows of the left matrix from 1M (2^{17}) to 8M (2^{20}). These suggest that **an RGNN system should be memory-efficient in order to accommodate larger models and datasets to fully utilize the massive resources on GPUs**. By eliminating unnecessary data copies, Hector achieves better memory efficiency than state-of-the-art systems.

The instruction per cycle (IPC) charts in Figure 12 indicate the traversal kernels are generally latency-bound: on RTX 3090, IPC is ideally 4 as each streaming multiprocessor (SM) has four schedulers. Backward propagation kernels have lower throughput due to worsened latency and increased memory bandwidth consumption by doubled memory accesses compared to forward propagation. In backward propagation, backward traversal kernels compute gradients using atomic updates, therefore hindering the throughput; GEMM kernels also on average have lower performance due to outer products that compute the delta of weights.

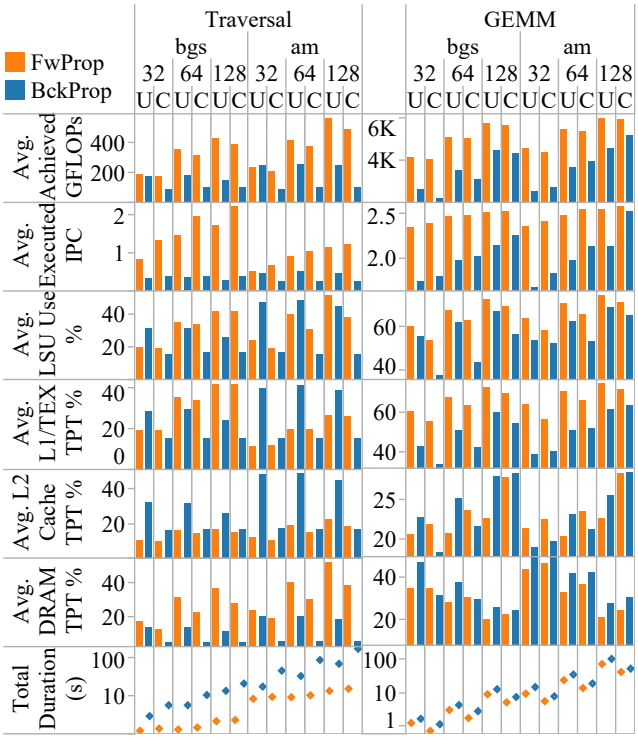


Figure 12. Architectural metrics of Hector kernels in the forward (Fw) and backward (Bck) propagation when running Hector on RGAT with compaction (C) and without (U). For each kernel category, aggregated duration and average (Avg.) metrics, e.g., instructions per cycle (IPC) and various throughputs (TPT), are reported.

5 Related Work

General GPU-accelerated GNN libraries. DGL [33] and PyG [7] are among the most popular GNN Python packages that enable easy development and evaluation of GNN models. DGL [33] proposes to implement GNN as SpMM/SDDMM operations. PyG's key scheme is scatter and gather operations that switch between edge-parallel regions and node-parallel regions. Hector instead built upon GEMM and traversal templates. By lowering the operators to GEMM as much as possible, Hector obtains better RGNN performance. Besides, DGL, PyG and work based on them do not currently provide inter-operator level IR. Our work shows the benefit of capturing inter-operator and inter-relation opportunities, e.g., linear-operator reordering, by operator rewrite at the inter-operator level IR. Systems without IR at this level eagerly execute operators without support for such optimizations.

GNN end-to-end compilers. Seastar [35] proposes a vertex-centric compiler stack to generate performant kernels throughout the training and/or inference of the model. Graphiler [36] proposes to program the message passing data flow graph and devises several TorchScript transforms to emit highly optimized inference code. Similarly, HGL [9] is an RGNN compiler. These prior arts 1) expose PyTorch tensors as operands of all operations to users and 2) replicate weight to unleash parallelism due to a lack of support for flexible data access schemes and/or code generation. Thus, they suffer more or less from memory inefficiency and performance degradation. Although the general concept of multi-level IR is not new, this work proposes new optimizations appropriate for each level and effective in reducing data movement and code bloat in the current state of practice: Linear operator reordering and compact materialization are two key and novel features to capture and eliminate repetitive computation across edge types. Section 3.7 discussed the generalizability of this work.

Kernel code optimization. FeatGraph [12] proposes code optimization framework on top of TVM [3] for user-defined-function-enabled SpMM and SDDMM. Some work proposed optimization for specific GNNs kernels. GE-SpMM [14, 15], and work [10] propose optimized schedules for SpMM. Others involve Seastar [35], PyTorch-Direct [23], and TLPGNN [8]. As our work shows, SpMM/SDDMM is not the only essential kernel in end-to-end RGNN execution. And our work is orthogonal to these prior arts as they can be incorporated into Hector as operator-specific schedules or new templates.

Code generation. SparseTIR [38] and TACO [18] propose IR and code generator for sparse tensor operations. MLIR [20] proposes multi-level IR design for deep learning. Aligned with this direction, FusedMM [30] unifies the SpMM and SDDMM CPU kernel. Hector is different as a higher-level compiler that optimizes the type of operators and generates efficient kernels to handle multiple edge/node types in the RGNN execution. SparseTIR and TACO are tensor-level compilers for sparse operators that may or may not specialize in deep learning. While we do not intend to reinvent the general-purpose sparse tensor code generator for completeness or performance, some of these works inspire us and may be incorporated to enhance the Hector code generator.

6 Discussion and Future Work

Extending the work to support for new optimizations. Hector is designed as an extensible framework to prototype

and evaluate new techniques. First, inter-operator optimizations can be prototyped as inter-operator level passes. Second, data layout optimizations can be supported by adding the corresponding intermediate data and adjacency access schemes discussed in Section 3.2. We plan to explore 1) if different sparse formats have any impact on the sparse kernel performance and 2) if there is any optimization opportunity in changing the intermediate data layout. E.g., storing edge attention and edge message together in one tensor where the innermost dimension becomes (size of hidden dimension + 1) would further reduce the number of kernels launched. Third, kernel optimizations can be prototyped as a kernel template and operator instances based on it. Alternatively, they can be implemented as operator-specific schedules.

Extending the work to use it in Distributed Systems. Due to the page limit, we focused this work on single-GPU performance. The kernels Hector generated could serve distributed systems, e.g., DistDGL [40]. Since performance improvement results from the reduction of data movements and memory footprints, it also applies to distributed systems.

Devise algorithms to select the layouts, optimizations, and schedules according to model, input graph, and GPU architecture. One of the most important compiler research problems is the algorithm that makes choices among the candidates in the design space. Apart from the input graph, the specific microarchitecture of each GPU model also makes a difference due to the architecture-specific features available, e.g., asynchronous loading to shared memory since Ampere [26], and different microarchitecture characteristics in each model. Therefore, it is meaningful to investigate their impact and incorporate them into decision making.

Optimize data movement in minibatch training. Graphs not fiting into GPU memory must stay in host memory or even storage during RGNN execution. In each step, the subgraphs are sampled and transferred to the GPU. With knowledge of graph semantics, data layout, and operator-specific schedules, Hector can help improve the scheduling of sampling and data transfer and generate CUDA kernels that gather data from the host memory on the fly [23].

Incorporate TACO to enhance the traversal code generation. In this work, we craft the code generators on our own for quick prototyping and focus on high-level optimizations. As this work establishes our understanding of what constructs are needed in the code generators for the traversal kernels, we think it viable to incorporate TACO for the code generation in the future because TACO provides a mature compiler infrastructure that enables the expression and application of optimizations [17] for sparse tensor operations in a principled manner, e.g., loop transformations. However, RGNN scenarios still pose several open challenges to TACO, especially in edge-centric operations. Take the edge-wise dot product when computing att_{HGT} in Figure 2 as an example. First, to balance the workload, we evenly split the edgewise loop and assign them to threading blocks. If we specify the source-node-wise loop and destination-node-wise loop as two dimensions in the TACO iteration space, we need to fuse these two loop levels to form the edgewise loop to split, but such loop fusion between two loop levels of iteration variables is not supported by TACO yet. Alternatively, we can specify the edgewise loop index as one dimension in the iteration space. In this case, we need indirect addressing to retrieve node data: We need to retrieve ① the source/destination node index by edgewise loop index and then ② the node data. However, indirect addressing is not natively supported in TACO and thus poses the second challenge.

7 Conclusion

RGNN execution has performance challenges due to the inherent computation pattern, the gap between the programming interface and kernel APIs, and the high kernel optimization cost due to the kernels' coupling with layout and heterogeneity. To systematically address these challenges, we presented the Hector IR and code generator for end-to-end RGNN training and inference. The IR design *decouples* the model semantics, data layout, and operator-specific schedule, and *expresses* these opportunities to allow them to be integrated into the design space as integral elements. Based on a GEMM template and a traversal template, Hector already achieves up to 43.7× speed-up in training and inference compared to state-of-the-art systems. Linear operator reordering and compact tensor materialization obtain up to 3.8× speedup compared to the Hector unoptimized code.

Acknowledgments

This work was partially supported by a grant from HPE, a gift from AMD/Xilinx, and equipment gifts from AMD/Xilinx and NVIDIA. The authors thank anonymous reviewers for their constructive comments. Kun owes thanks to Dr. Seung Won Min and Prof. Guohao Dai for initiating this collaboration and precious advice on this project. The authors thank Prof. Deming Chen, Dr. Penporn Koanantakool, Dr. Dejan Milojicic, and Prof. Vikram Adve for hosting meetings and seminars where many constructive feedbacks were given.

References

[1] Stephan Bloehdorn and York Sure. Kernel Methods for Mining Instance Data in Ontologies. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Karl Aberer, Key-Sun Choi, Natasha Noy, Dean Allemang, Kyung-Il Lee, Lyndon Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web*, volume 4825, pages 58–71. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. http://link.springer.com/10.1007/978-3-540-76298-0_5.

[2] Dan Busbridge, Dane Sherburn, Pietro Cavallo, and Nils Y. Hammerla. Relational graph attention networks. *arXiv preprint arXiv:1904.05811*, 2019. https://arxiv.org/abs/1904.05811.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018. https://www.usenix.org/conference/osdi18/presentation/chen.

[4] Victor de Boer, Jan Wielemaker, Judith van Gent, Michiel Hildebrand, Antoine Isaac, Jacco van Ossenbruggen, and Guus Schreiber. Supporting Linked Data Production for Cultural Heritage Institutes: The Amsterdam Museum Case Study. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Elena Simperl, Philipp Cimiano, Axel Polleres, Oscar Corcho, and Valentina Presutti, editors, *The Semantic Web: Research and Applications*, volume 7295, pages 733–747. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. http://link.springer.com/10.1007/978-3-642-30284-8_56.

[5] Gerben K. D. de Vries. A Fast Approximation of the Weisfeiler-Lehman Graph Kernel for RDF Data. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Camille Salinesi, Moira C. Norrie, and Óscar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908, pages 606–621. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. http://link.springer.com/10.1007/978-3-642-40988-2_39.

[6] Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, February 1991. https://pubs.acs.org/doi/abs/10.1021/jm00106a046.

[7] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019. https://arxiv.org/abs/1903.02428.

[8] Qiang Fu, Yuede Ji, and H. Howie Huang. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, pages 122–134, New York, NY, USA, June 2022. Association for Computing Machinery. https://doi.org/10.1145/3502181.3531467.

[9] Yuntao Gui, Yidi Wu, Han Yang, Tatiana Jin, Boyang Li, Qihui Zhou, James Cheng, and Fan Yu. HGL: Accelerating Heterogeneous GNN Training with Holistic Representation and Optimization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2022. https://dl.acm.org/doi/abs/10.5555/3571885.3571980.

[10] Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei Hwu. At-scale sparse deep neural network inference with efficient gpu implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020. https://doi.org/10.1109/HPEC43674.2020.9286206.

[11] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. http://arxiv.org/abs/2005.00687, February 2021.

[12] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, Atlanta, GA, USA, November 2020. IEEE. https://ieeexplore.ieee.org/document/9355318/.

[13] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous Graph Transformer. In *Proceedings of The Web Conference 2020*, pages 2704–2710, Taipei Taiwan, April 2020. ACM. https://dl.acm.org/doi/10.1145/3366423.3380027.

[14] Guyue Huang, Guohao Dai, Yu Wang, Yufei Ding, and Yuan Xie. Efficient Sparse Matrix Kernels based on Adaptive Workload-Balancing and Parallel-Reduction. http://arxiv.org/abs/2106.16064, October 2021.

[15] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. GE-SpMM: General-Purpose Sparse Matrix-Matrix Multiplication on GPUs for Graph Neural Networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, November 2020. https://dl.acm.org/doi/10.5555/3433701.3433796.

[16] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016. https://arxiv.org/abs/1609.02907.

[17] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 180–192, Washington, DC, USA, February 2019. IEEE. https://doi.org/10.1109/CGO.2019.8661185.

[18] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, October 2017. https://dl.acm.org/doi/10.1145/3133901.

[19] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 1–6, New York, NY, USA, November 2015. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/2833157.2833162.

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Arnaud Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO 2021*, 2021. https://ieeexplore.ieee.org/document/9370308.

[21] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4013–4021, 2016.

[22] Wen mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors*. Morgan Kaufmann, fourth edition, 2023. https://www.sciencedirect.com/book/9780323912310/programming-massively-parallel-processors.

[23] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. Large graph convolutional network training with GPU-oriented data communication architecture. *Proceedings of the VLDB Endowment*, 14(11):2087–2100, July 2021. https://dl.acm.org/doi/10.14778/3476249.3476264.

[24] Israt Nisa. [Feature] Gather mm by isratnisa · Pull Request #3641 · dmlc/dgl. https://github.com/dmlc/dgl/pull/3641, January 2022.

[25] Israt Nisa, Minjie Wang, Da Zheng, Qiang Fu, Ümit Çatalyürek, and George Karypis. Optimizing irregular dense operators of heterogeneous gnn models on gpu. In *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 199–206, 2023. https://doi.org/10.1109/IPDPSW59300.2023.00042.

[26] Nvidia. Controlling Data Movement to Boost Performance on the NVIDIA Ampere Architecture. https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/, September 2020.

[27] Nvidia. Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores | NVIDIA Technical Blog. https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/, March 2021.

[28] Nvidia. cublas<t>gemmBatched() | cuBLAS Library User Guide v12.2. https://docs.nvidia.com/cuda/cublas/index.html#cublas-t-gemmbatched#:~:text=make%20multiple%20calls%20to%20cublas%3Ct%3Egemm, July 2023.

[29] PyTorch. TorchScript — PyTorch 2.2 documentation. https://pytorch.org/docs/stable/jit.html, January 2024.

[30] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 256–266, May 2021. https://ieeexplore.ieee.org/document/9460486.

[31] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling Relational Data with Graph Convolutional Networks. In Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam, editors, *The Semantic Web*, volume 10843, pages 593–607. Springer International Publishing, Cham, 2018. http://link.springer.com/10.1007/978-3-319-93417-4_38.

[32] Kristina Toutanova and Danqi Chen. Observed versus latent features for knowledge base and text inference. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and Their Compositionality*, pages 57–66, Beijing, China, July 2015. Association for Computational Linguistics. https://aclanthology.org/W15-4007.

[33] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019. https://arxiv.org/abs/1909.01315.

[34] Zhaokang Wang, Yunpan Wang, Chunfeng Yuan, Rong Gu, and Yihua Huang. Empirical analysis of performance bottlenecks in graph neural network training and inference with GPUs. *Neurocomputing*, 446:165–191, July 2021. https://www.sciencedirect.com/science/article/pii/S0925231221003659.

[35] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. Seastar: Vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 359–375, Online Event United Kingdom, April 2021. ACM. https://dl.acm.org/doi/10.1145/3447786.3456247.

[36] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. Graphiler: Optimizing graph neural networks with message passing data flow graph. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 515–528, 2022. https://proceedings.mlsys.org/paper/2022/file/a87ff679a2f3e71d9181a67b7542122c-Paper.pdf.

[37] Zhiqiang Xie and Zihao Ye. Graphiler Repository on Github. https://github.com/xiezhq-hermann/graphiler, January 2023.

[38] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3582016.3582047.

[39] Da Zheng and George Karypis. The Nature of Graph Neural Network Workloads. https://hc33.hotchips.org/assets/program/tutorials/HC2021.Amazon.DaZheng.v2.pdf, August 2021.

[40] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. In *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pages 36–44, GA, USA, November 2020. IEEE. https://doi.org/10.1109/IA351965.2020.00011.