RNDN: OPTIMIZED QUERY COMPILATION FOR GPUS

by
Alexander Krolik

School of Computer Science
McGill University, Montreal

May 2022

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

To Laurie

Abstract

Ever expanding data necessitates efficient analysis, with query compilation proving an effective optimization technique. GPU database systems complement this approach, exploiting a parallel accelerator increasingly available on consumer devices. They are limited, however, by the high costs of data transfer and compilation that reduce performance on workloads with varying data and queries. In addition, many such existing systems fail to execute comprehensive benchmark sets. We present a new approach to query compilation for GPUs, providing an optimized compilation pipeline that significantly increases performance on end-to-end evaluation. Trading minor slowdowns in execution for major speedups in compilation, we replace the expensive proprietary pipeline by our own runtime-suitable compiler and assembler that specifically target relational queries. In particular, we take SQL queries written in *HorseIR*, outline kernels of compatible operations, and produce a parallel intermediate representation. This *PTX* code is translated to machine instructions and assembled to binary for execution on the CUDA platform. Each operator is implemented by a simplistic yet efficient algorithm, and complemented by an optimized execution engine. Compared to existing CPU and GPU database systems, we show performance improvements on both cached and end-to-end workloads while maintaining completeness on a significant benchmark suite. Importantly, we effectively trade execution for compilation, significantly outperforming proprietary compilation pipelines without excessively degrading cached query performance.

Résumé

L'augmentation des données nécessite une analyse efficace et la compilation des requêtes est une des techniques d'optimisation convenable pour la performance. Les systèmes de base de données à processeurs graphiques agrémentent cette technique en exploitant l'accélérateur parallèle fréquemment offert sur les appareils des consommateurs. Toutefois, ils sont limités par les coûts de transfert et de compilation élevés qui réduisent la performance lorsque les requêtes ou les données sont variables. De plus, plusieurs de ces systèmes de base de données à processeurs graphiques ne parviennent pas à exécuter des tests de performance complets. Alors, nous présentons une nouvelle approche à compilation de requêtes SQL, pour les processeurs graphiques, en fournissant un pipeline de compilation optimisé, qui améliore considérablement la performance des évaluations de bout en bout. Dans l'approche utilisée, nous remplaçons le pipeline propriétaire coûteux par un compilateur et un assembleur adaptés à l'exécution, qui ciblent les requêtes relationnelles. Dans l'ensemble, pour notre système, des ralentissements mineurs lors de l'exécution seront remplacés par des accélérations majeures dans la compilation. En particulier, nous prenons des requêtes SQL écrites en *HorseIR*, nous définissons les noyaux de calcul d'opérations compatibles et nous produisons un langage intermédiaire parallèle. Ce langage intermédiaire parallèle *PTX* est traduit en code machine et il est assemblé en binaire pour être exécuté sur la plateforme CUDA. Chaque opérateur est mis en œuvre par un algorithme simpliste et pourtant efficace, et il est soutenu par un moteur d'exécution optimisé. Contrairement aux systèmes de base de données CPU et GPU existants, nous démontrons des améliorations dans les résultats, tout en maintenant l'exhaustivité sur un ensemble représentatif de requêtes SQL, sur les tests de performance de requêtes stockées et de

bout en bout. En somme, notre système sacrifie l'exécution pour l'amélioration de la compilation et la performance de bout en bout. Elle surpasse de manière importante les pipelines de compilation propriétaires sans dégrader excessivement la performance des requêtes stockées.

Acknowledgements

Firstly, I would like to thank my supervisors, Clark Verbrugge and Laurie Hendren, for all of their support since way back in my undergrad. From my first summer research project to this thesis, I have always appreciated your guidance and encouragement to try new ideas and tackle the next problem – all while lending an ear to whatever obstacles I encountered, no matter where I faced them. Even though Laurie sadly will never see this complete thesis, her influence has been present throughout this process and will stay with me for the years ahead. On a personal note, I miss you, and will never forget your curiosity, determination and compassion. And a special thank you to Clark for helping with the transition and for encouraging me to extend this work well beyond the initial scope and into new areas. While not a direct supervisor, I would also like to thank Bettina Kemme for the inspiration in she provided in her research course that later became a key part of this thesis.

I would also like to thank my lab mates who I have worked with and been friends with over the years. Whether that be writing papers and developing solutions with Hanfeng Chen, lengthy discussions of ideas or challenges with Prabhjot Sandhu and David Herrera, bouncing my thoughts around with Erick Lavoie, Akshay Gopalakrishnan, or Steven Thephsourinthone, or the many many others who I have met over the years, you have all been part of this work in one way or another. I also really appreciate the warmth that we have as a group, and the encouragement and kindness as we navigated this chapter together.

I also want to thank my family and friends who have listened and encouraged me to pursue my goals, and helped me along the way. In particular, Dominique Ferland for her help translating the French abstract, Lei Lopez for her friendship since my

first summer project, Giulia Alberini her teaching camaraderie, Vincent Foley and Antonio Giordano for their help with COMP 520, and Kamil Legault and Ziuwin Leung for listening in the evenings to the challenges of the day. I would also like to give a very special thank you to my best friend and companion, Alexander Patton, for his caring support and kindness.

I would also like to give thanks to the School of Computer Science, its then-director Bettina Kemme, and most of all Laurie Hendren for their trust in teaching COMP 520 (Compiler Design). It was an amazing experience, and has been a key part of learning how to present ideas to a new audience. Thank you to all the students I taught over the years for your patience and enthusiasm, to my TAs for their help and support with teaching, and to the all administrative and technical support staff in the department for your help making it happen.

I would like to acknowledge the previous efforts to explore NVIDIA architecture, and in particular the work by Scott Gray on the Maxwell and Pascal architectures. Their projects and documentation were instrumental to this thesis, and laid the groundwork required for many of our techniques.

Laslty, I would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Laurie Henden and Clark Verbrugge, the School of Computer Science and McGill University for their funding to complete this thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

AST: Abstract syntax tree

CFG: Control-flow graph

GPU: Graphics processing unit

GPGPU : General-purpose graphics processing unit

ILP: Instruction-level parallelism

IR: Intermediate representation

JIT: Just-in-time

PC: Program counter

PE: Processing element

PTX: Parallel thread execution

RDBMS: Relational database management system

SASS: Shader assembler

SIMD: Single-instruction multiple-data

SM: Streaming multiprocessor

Chapter 1
Introduction

With sizes ever increasing, the efficient storage and analysis of large scale data is a fundamental question for the database community and has driven significant research in recent years. Compilation is one effective proposal for repeated queries, eliminating the interpretation overhead found in traditional designs and amortizing the additional cost over multiple executions [38, 150, 152]. More recently, database-specific approaches have also been used in place of general-purpose compilers, allowing competitive performance over interpretation even for new queries by performing only necessary optimization [69, 117]. Alternatively, expensive computation can be offloaded to specialized accelerators like the GPU, exploiting their vast parallelism and high memory bandwidth for impressive speedup [150, 103, 20]. Unfortunately, despite widespread adoption in modern architectures, their use has so far been hindered by the low memory capacity and slow transfer speeds of current hardware, the high compilation cost of current pipelines, and the complex design of efficient parallel GPU algorithms. Existing approaches are therefore best suited to contexts where data can be completely cached in device memory and the query either pre-compiled or interpreted. Additionally, current systems typically support only a subset of database queries [45], and require low-level expertise for algorithm design due to their unconventional and sometimes closed-specification architecture. High performance GPU databases are thus within reach, but their their potential is still limited compared their well-researched CPU counterparts in many scenarios.

In this thesis we propose rNdN, an end-to-end and open-source[1] compilation pipeline and execution engine for relational database queries on the GPU. Our approach is compiler-first, focusing on balanced and runtime-suited compilation strategies rather than algorithm design and optimization. This removes an important limitation faced by existing work and extends the use of compiled GPU databases to short running queries and competitive end-to-end evaluation. We also aim for simplicity, composing intuitive techniques and algorithms rather than intensive query- or domain-specific optimization. Despite being low-level, we maintain significant generality and can adapt to future architectures. We compare our approach to modern and open-source CPU and GPU database systems, both interpreted and compiled [101, 38, 150, 103]. Our results show performance improvements in several key use cases, notably *cached* and *end-to-end* scenarios, over all comparison systems and on a variety of queries. Compilation speedups exceed **5.5x** geomean, leading to end-to-end improvements of **4.4x** compared to NVIDIA's optimizing pipeline for Ampere. Importantly, we also support a complete analytics benchmark. We present research challenges in Section 1.1, specific contributions of this thesis in Section 1.2, related publications in Section 1.3, and the thesis roadmap in Section 1.4.

1.1 Challenges

Our approach targets short running SQL queries with smaller yet significant data sizes, improving performance through GPU acceleration without the usual compilation overhead. This presents the following challenges, that we address in our design:

Extracting parallelism: Query programs define the execution from input to output, comprising a mix of data organization and computation. To efficiently exploit the GPU, we must therefore extract parallel sections from arbitrary programs, each corresponding to a kernel. For best performance, parallelism must be maximized while the cost of data produced by each operation minimized.

[1] https://github.com/akrolik/rNdN

Exploiting hardware: Most GPU algorithms are designed for a narrow problem and require significant manual and low-level optimization for best performance. As a runtime compiler approach, we must therefore define efficient code generation patterns for a wide variety of functionality without requiring extensive optimization. Additionally, since NVIDIA hardware details are relatively opaque, extensive analysis is required to best exploit underlying hardware capabilities.

Data management: Due to well-known size limitations of GPU memory and PCI-e bandwidth, proper data transfer and allocation strategies are required. This is especially relevant as database systems manipulate large amounts of data.

Fast compilation time: As end-to-end query execution includes the compilation cost, an effective implementation must offset additional overhead with improvements in computation. For short running queries in particular, the compilation overhead is many times larger than the computation itself – on average over **75%** of the overall execution. This requires addressing a major shortcoming of existing GPU compilers, namely their high cost, without slowing computation.

1.2 Contributions

rNdN is complete database system that compiles and executes SQL queries on the GPU. We divide contributions into 3 categories discussed below: (1) the compiler and assembler pipeline; (2) a supporting runtime; and (3) performance evaluation.

Compilation Pipeline

Kernel outliner: Using shape and geometry abstractions that capture GPU parallelism, and the implicit synchronization defined in array-based intermediate representations, we automatically pipeline compatible operations into efficient kernels beyond simple element-wise functions. Given an arbitrary query program in HorseIR, we propose an efficient analysis chain that extracts parallelism and minimizes costly intermediate data materialization.

High-level code generation: Lowering to a GPU-specific intermediate representation, PTX, we bypass expensive high-level language compilers while maintaining high performance. We select efficient code generation strategies that adapt existing, simplistic algorithms to execute database queries written in HorseIR.

Low-level code generation: As the cost of assembling CUDA-compliant binaries is excessive for runtime use, we propose an alternative assembly pipeline that trades minor slowdowns in execution for major speedups in compilation. Extending the existing reverse-engineering efforts and runtime compiler research, we explore and formalize the necessary components for generating high performance code for multiple architectures without the usual compilation overhead.

End-to-end pipeline: To our knowledge, we present the first detailed, end-to-end and open-source solution for generating an efficient CUDA binary from a high-level language without the use of proprietary or closed-source components.

Runtime

Supporting compiled queries on the GPU, we formalize the necessary steps for the execution engine, present a mapping abstract kernel geometries to thread layouts, and motivate the necessary data management optimizations for GPU buffers.

Coverage and Performance

Coverage: We correctly execute all 22 queries of the widely used TPC-H benchmark, supporting and accelerating a variety of database computation. While we are not the first GPU database to achieve this feat, none of the recent systems we evaluated offered complete support for the entire benchmark.

Performance: Our system shows significant improvement over both CPU and GPU database systems, both compiled and interpreted, and on a variety of use cases. Additionally, we demonstrate that our approach effectively balances compilation and execution for significant speedup in end-to-end execution.

Evaluation: We present a complete execution breakdown of the compilation and computation phases, identify the key trade-offs and performance implications, and enable reasoned optimization of GPU databases.

1.3 Publications

Portions of this thesis have been previously published, and their approach integrated into our overall system. We note two publications related to the design of database systems, and two miscellaneous publications on compilers and parallelism.

1. **Alexander Krolik**, Clark Verbrugge, and Laurie Hendren. r3d3: Optimized query compilation on GPUs. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 277–288, 2021. [125]

   Contributions: This paper presents the initial system design of rNdN, and concerns the frontend compilation and execution of database queries on the GPU. I proposed the design, implementation and evaluation of the r3d3 system.

2. Hanfeng Chen, **Alexander Krolik**, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. Improving database query performance with automatic fusion. In International Conference on Compiler Construction, pages 63–73, 2020. [39]

   Contributions: Developed concurrently with our GPU approach, this paper presents CPU loop fusion techniques for HorseIR queries. I assisted with writing and discussed ideas for the HorseIR specification and shape analysis/fusion.

Miscellaneous Publications

1. Hanfeng Chen, **Alexander Krolik**, Erick Lavoie, and Laurie Hendren. Automatic vectorization for MATLAB. In Languages and Compilers for Parallel Computing, pages 171–187, 2017. [40]

   Contributions: Addressing vectorization of MATLAB programs, this work proposes an algorithm for analyzing and transforming loops. I assisted with presentation, organization and writing.

2. Clark Verbrugge, Christopher J. F. Pickett, **Alexander Krolik**, and Allan Kielstra. Exhaustive analysis of thread-level speculation. In 3rd International Workshop on Software Engineering for Parallel Systems, page 25–34, 2016. [226]

   Contributions: This work proposes strategies for launching and joining speculative execution of loop iterations. I conducted scalability experiments and addressed correctness of the dynamic programming algorithm.

1.4 Roadmap

The remainder of this thesis is organized as follows:

Chapter 2: We provide background on database systems, the HorseIR intermediate representation, and GPU architectures and compilation.

Chapter 3: We describe a high-level overview of our system design and the components required to compile and execute database queries on the GPU.

Chapter 4: We define the frontend and compiler pipeline that outlines pipelined kernels and generates efficient PTX intermediate code for database queries.

Chapter 5: We define the backend compiler (assembler) pipeline that translates from intermediate form to machine code. We discuss low-level details including register allocation, code templates, instruction scheduling, and binary files.

Chapter 6: We outline the runtime system that supports execution of the compiled queries, including the execution engine and data management.

Chapter 7: We evaluate the performance of our system, breaking down the importance of each phase, comparing against modern CPU and GPU databases, and considering the impact of each optimization.

Chapter 8: We contextualize our work, presenting other research in compiled queries, GPU databases, compilers, and assemblers.

Chapter 9: We conclude with important takeaways and propose future research that could extend the use of our system beyond database queries.

Chapter 2
Background

In this chapter we present the background for rNdN, detailing the basis for our work as well as the specialized parallel hardware used in our approach. We begin by briefly describing relational database systems and SQL query processing in Section 2.1. Next, we present our chosen representation of SQL queries, HorseIR, and its implications in Section 2.2. Finally, we explore GPUs in Section 2.3, including high and low-level architecture views, programming, and compilation pipelines.

2.1 Databases

With data sizes ever growing, efficient storage and analysis has become increasingly important. To this end, relational database systems (RDMBS) are optimized implementations that provide tabular data storage and "powerful query languages" [188]. Based on the relational model, data is either stored by row (row-oriented) or by column (column-oriented), with the latter designed for efficient querying [214]. Data may also be stored on-disk or in main-memory, removing the I/O bottleneck and increasing performance [101]. We focus on in-memory, column-oriented databases in our approach as they are more easily adapted to GPUs. The query language is described in Section 2.1.1, and execution in Section 2.1.2.

2.1.1 SQL

SQL is a declarative programming language used for querying relational databases, uncoupling the output from its implementation. Based on relational algebra, it comprises a set of operations that analyze relational data. Common query operations are summarized below, with more details available in other resources [188].

Project Select a subset of columns.

Where Select a subset of rows, according to a boolean predicate.

Aggregate Equivalent to reduction, combine data into a single value (e.g. `SUM`, `MIN`).

Join Combine data from related tables using a cross product.

Group Collect elements with common values into bins, each of which is aggregated.

Sort Orders rows according to one or more attributes.

Distinct Select only unique values.

Each operation varies in cost, depending on its computation and the underlying data. Joining and grouping are the most expensive, while projection and selection are comparably lightweight. A simple example query is shown in Figure 2.1 that selects a subset of country data according to a boolean predicate.

```
SELECT name
FROM country
WHERE size > 10
```

Figure 2.1 – SQL query projecting (`SELECT`) the name attribute of all countries `FROM` the county table `WHERE` the size attribute is greater than 10.

2.1.2 Execution

SQL queries are translated to a relational algebra tree, representing the required operations as well as their dependencies [33]. The tree is optimized to find a minimal cost *execution plan*, with each operation augmented with algorithmic details. During optimization, multiple plans are produced and the most efficient is heuristically selected [188]. The query plan can then either be evaluated in an interpreter using efficient operator implementations [101], or compiled to binary for execution using additional optimizations like pipelining [152, 39]. Compiling the query typically yields faster execution, but must be balanced against compilation time for overall performance in cold-start, uncached environments. Compilation overhead can be amortized by repeated execution, or reduced using caching or minimal optimization. In this project we use execution plans from HyPer, a CPU database system designed for efficient execution and compilation [152].

2.2 HorseIR

HorseIR is an array-based intermediate representation (IR) designed to represent and optimize relational database queries [38]. Given an SQL query, the database system produces an efficient query plan with SQL-specific heuristics, translates the plan to HorseIR, and applies traditional compiler optimizations. The database system can therefore combine benefit from both the compiler and database fields. Notably, as an array-based language, it is ideal for vector processing or SIMD architectures.

The language provides a comprehensive set of built-in functions from array-based languages, including unary and binary functions (e.g. `@abs`, `@plus`) and algebraic operations (e.g. boolean selection `@compress`). Queries are represented using a mixture of these basic functions and SQL-specific operations. For example, general purpose boolean compression supports `SELECT`, while complex operations like `GROUP` and `JOIN` have dedicated built-in functions (e.g. `@group` and `@join_index`). Similarly, the type system comprises basic *vector* types such as `i32`, `f64` and `str` (32-bit integer, 64-bit float and string) as well as database-specific formats. Vectors are analogous

to column data, while *lists* represent tuples of columns. Lists may either be homogeneous or heterogeneous depending on the column types. Built-in functions `@each*` operate on lists directly, applying array-based functions to each cell independently. *Enumerations* designate foreign keys similar to pointers, and *tables* and *dictionaries* support tabular data and key-value mappings respectively. Compound types are parameterized by their elements, simplifying code generation and optimization.

An example HorseIR program is shown in Figure 2.2, implementing the prior SQL query shown in Figure 2.1. Recall that the query selects a subset of countries according to a boolean size predicate. In the first phase, table and column data is loaded from the underlying store, with each column represented as a vector. It must then be cast to the appropriate type for the query, ensuring correctness of the subsequent code. The predicate is then evaluated, returning a vector of `true/false` values, one for each row. Using this *boolean mask*, compression selects a subset of the input data, extracting only those rows corresponding to a `true` value. Finally, the result is formed into a table and returned to the user. Note that tables are represented as a vector of *symbol* column names (`sym`), and an equivalently sized list of columns. As a compound type, lists can either be parameterized by their element types, or specify the wildcard type `?` that is inferred by the compiler.

```
def main() : table {
    // Load table and columns
    var t1:table   = @load_table(`country:sym);
    var t2:i32     = check_cast(@column_value(t1,`size:sym),i32);
    var t3:sym     = check_cast(@column_value(t1,`name:sym),sym);
    // Evaluate selection predicate amd select only true rows
    var t4:bool    = @gt(t2,10:i32);
    var t5:sym     = @compress(t3,t4);
    // Form output table
    var t6:list<?> = @list(t5);
    var t7:table   = @table(`name:sym,t6);
    return t7;
}
```

Figure 2.2 – SQL query represented in HorseIR. The program loads column data, selects rows with predicate (`size > 10`), and forms the output table.

2.3 GPUs

A modern GPU is a multi-core execution unit designed for general throughput computation (GPGPU) [136]. Compared to the traditional CPU-system, the architecture is thus fundamentally optimized for parallel applications and presents unique design and implementation challenges. We focus on NVIDIA GPUs in this work, although other vendors all follow similar overall designs. We explore the a high-level abstract architecture in Section 2.3.1, two recent hardware implementations in Section 2.3.2, before discussing compiler design in Section 2.3.3.

2.3.1 Abstract Architecture

At the most granular level, an abstract GPU consists of a large set of *processing elements* (PE), each conceptually representing a core of the execution unit. Organized hierarchically, processing elements are grouped into *streaming multiprocessors* (SM), and the set of multiprocessors form the GPU as shown in Figure 2.3. Compared to a CPU, there are thus a larger number of cores, though as the cores are simpler in design and relatively slower, the GPU is most amenable to throughput computation. Additionally, due to the hierarchical design, algorithms must judiciously use synchronization for efficient execution. While cores within a multiprocessor do have limited synchronization and data-sharing capabilities, full inter-multiprocessor synchronization is not supported by current architectures.

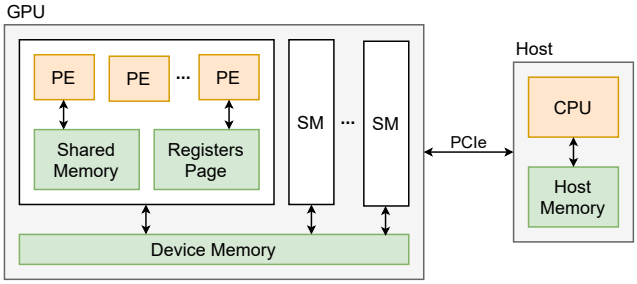


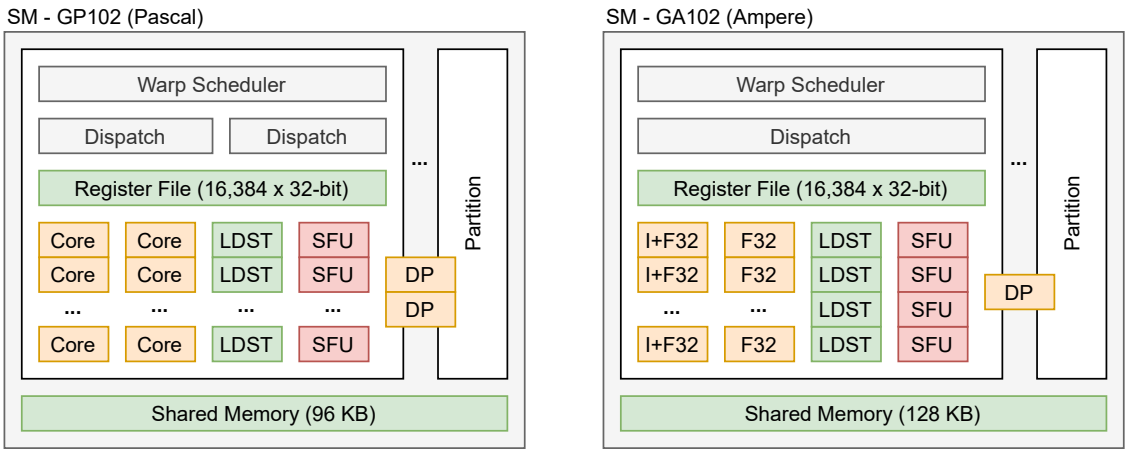
Figure 2.3 – Abstract GPU architecture. (Adapted, © 2021 IEEE [125])

Memory follows a similar pattern, with a distinct store for each level in the hierarchy. Each processing element is assigned a segment of the *registers page*, multiprocessors have fixed-size *shared memory*, and *device memory* is universally accessible. In data-heavy programs, memory requirements for registers and shared memory thus limit the amount of parallelism. Device memory also serves as the entry and exit point for data transferred over the PCIe-bus from the host system. Latency follows capacity, with smaller stores having correspondingly lower access times, while device memory is the largest and highest latency. Appropriate data caching is thus important to improve performance.

A GPU program specifies the behaviour of a single thread, as well as any inter-thread synchronization. Threads are organized into *thread blocks* which in turn form the *grid*. At runtime, thread blocks are dynamically assigned to a multiprocessor wherein threads are dispatched to cores in groups of 32 (a *warp*). The number of concurrently executing thread blocks is therefore limited by the underlying hardware and precludes full inter-block synchronization. Additionally, despite the conventional wisdom that all threads within a block execute in lock-step, as the number of threads within a block greatly exceeds the number of cores, synchronization is required for consistent memory. GPUs thus provide intra-block thread barriers, and global and shared atomics. Efficient GPU programs must therefore saturate cores with independent threads and effectively use synchronization. Note that as thread-creation overhead is low, increased parallelism can yield overall speedup.

2.3.2 Real Architecture

A high-level view of GPU architectures is sufficient for most applications, allowing developers to adapt their algorithms. However, intensive optimization and machine code generation require a lower-level understanding as the underlying hardware has additional characteristics not exposed in user-level code. Two recent NVIDIA architectures, Pascal (2016) [158] and Ampere (2020) [160], are shown in Figure 2.4 and explored and compared below. In this thesis, we target both architectures using a general-purpose approach, showing the utility across platforms.



(a) NVIDIA GTX 1080 Ti (Pascal) [158, 222].

(b) NVIDIA GTX 3080 (Ampere) [160].

Figure 2.4 – Comparison of recent consumer NVIDIA GPU architectures.

A GPU multiprocessor is divided into partitions, each executing a collection of *warps*. A warp is a set of 32 consecutive threads, all belonging to the same thread block. During execution, thread blocks are assigned to multiprocessors and warps divided among partitions according to a hardware-specific allocation scheme[1]. Each clock cycle, the *warp scheduler* selects the next available warp and dispatches the current instruction to the appropriate *functional unit*. In the case of dual issue, 2 consecutive instructions may be issued per warp, as discussed in Chapter 5. Functional units commonly include cores, special function units (SFU; e.g. cosine, sine, reciprocal) and double-precision units for computation (DP), as well as load-store units (LDST) for data accesses. Cores may either be general-purpose for both integer and single-precision computation (Pascal) or more specialized as in Ampere. Note that some units like double-precision may be shared between multiple partitions [222]. If the number of functional units is less than the size of a warp, the throughput is limited and multiple cycles are needed to issue the instruction [168]. The published number of CUDA cores corresponds to the number of single-precision floating point

[1] https://forums.developer.nvidia.com/t/some-doubts-about-the-task-scheduling-of-nvidia-gpu/50263

Table 2.1 – Comparison of Pascal and Ampere GPU properties [158, 222, 160].

Property		**Pascal (1080 Ti)**	**Ampere (3080)**
# Multiprocessors		28	68
# Partitions		4	
# Total Cores		3584	8704
Dispatch		Dual	Single
Memory	Device Memory	11 GB	10 GB
	Shared Memory	96 KB	128 KB
	Registers	16,384 x 32-bit	
Functional Units	Cores	32 x I32/F32	16 x I32/F32; 16 x F32
	LDST	8	4
	SFU	8	4
	DP	2 shared	1 shared

units. A comparison of the functional units and partition properties of both architectures is shown in Table 2.1. Note that shared memory is multiprocessor-wide, whereas registers are allocated per partition. At first glance the newer architecture has a smaller number of functional units and limited dispatch. However, combining the increased number of multiprocessors with more efficient instructions leads to performance improvements. Further architecture-specific details are described in later chapters as we explore the compiler implementation.

2.3.3 Compilation Pipeline

Compilation for NVIDIA GPUs follows a 2 stage process, transforming a high-level SIMT program to a binary form executable on the hardware as shown in Figure 2.5 [167]. In the first stage, user-written CUDA code is translated to PTX, a low-level intermediate representation designed by NVIDIA targeting a virtual architecture [170]. This avoids complications from register allocation, instruction scheduling, and binary formats which all depend on hardware implementations. The PTX code can then be distributed to user machines where it will undergo stage 2 compilation. As the intermediate code is mostly architecture independent, delaying compilation has the advantage of supporting multiple systems with a single implementation, in addition to being faster to compile than the original code. Note that PTX is a

versioned IR, which has different functionality depending on the *compute capability*, an abstract versioning system for GPU architecture. The distributed code must therefore only use features supported by the target system and compute capability.

During stage 2, the compiler (also referred to as the assembler) translates from PTX to SASS, the machine code used in the CUDA binary format (`.cubin`) [162]. SASS is short for "**S**treaming **Ass**embler"[2], and is tied to a specific GPU family. As the architecture evolves, the underlying instruction set changes, including both the binary format and operations supported. The assembler must therefore consider properties of the real architecture in its implementation, constraints not present in the high-level program. As it executes on the target machine, it should also be efficient or use caching to reduce overhead.

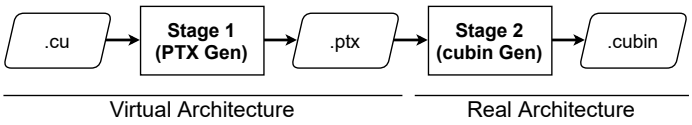


Figure 2.5 – NVIDIA CUDA compilation pipeline [167].

[2] https://stackoverflow.com/questions/9798258/what-is-sass-short-for

Chapter 3
Overview

rNdN is a complete GPU compiler and execution engine specializing on SQL database queries. Given a HorseIR query program, it extracts parallel kernels (Frontend), compiles to binary (Compiler+Assembler), executes the program and returns the result (Runtime). We emphasize end-to-end performance, dependent on runtime-efficient compilation and optimized execution. The design is separated into 3 main components shown in Figure 3.1: Frontend and Compiler, Assembler, and Runtime described in Section 3.1, Section 3.2 and Section 3.3 respectively.

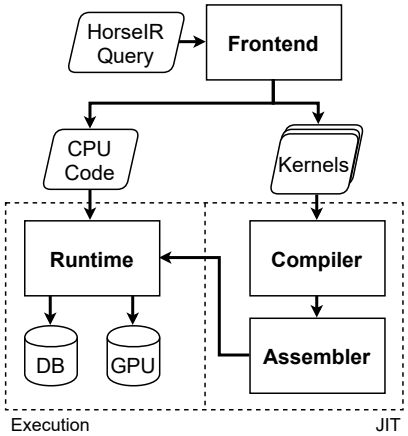


Figure 3.1 – rNdN overall system architecture.

3.1 Frontend and Compiler

We follow the standard 2 stage compilation pipeline, beginning with our frontend. The first stage compiler parses the input query, decomposes the program into kernels and CPU control code, and produces GPU code for the parallel sections as shown in Figure 3.2. This separates the problem of parallelization from the complexities of code generation and execution, and allows selectively offloading expensive computation. We can thus effectively improve performance over typical CPU database systems.

A query program consists of a single function entry point, calling a sequence of built-in functions. We begin by parsing the HorseIR program, and enforcing semantic requirements. As a statically-typed language, this includes building a symbol table and type-checking before ensuring that all variables are assigned before use. Since we target a typed low-level IR, static-typing is imperative for high-performance.

Given the parsed program, we then determine parallel sections using the Outliner, so-called because it "outlines" sections of the original program that are suitable for the GPU. This approach was inspired by the Velociraptor compiler which similarly extracts *interesting regions* [74]. The remaining program consists of CPU control code and sequential operations. Kernels are designed to maximize parallelism and minimize high-latency data accesses, while satisfying dependencies and synchronization.

Each kernel can then be compiled to PTX code, mapping from the high-level built-in functions to a parallel GPU equivalent. As outlined kernels are valid sequences of operations, our generation strategy is simple and efficient, keeping compilation overhead low. In addition, optimizations necessary for performance are embedded directly in the code patterns instead of a separate phase. A full description of the outliner and code generation strategies is discussed in Chapter 4.

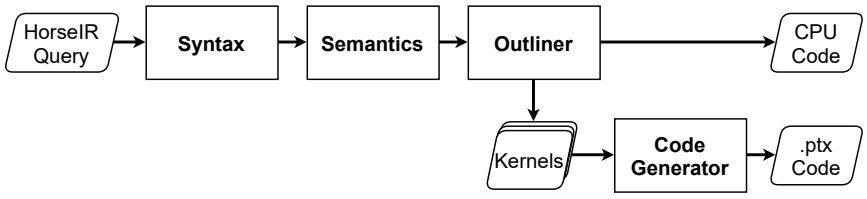


Figure 3.2 – rNdN frontend compiler architecture.

3.2 Assembler

PTX code is typically assembled to binary by time-costly NVIDIA proprietary tools, providing limited insight and control into the process. We present a drop-in replacement assembler that translates PTX code to SASS machine code while minimizing overhead as shown in Figure 3.3. This enables runtime JIT compilation previously unsuitable for short-running programs typical for query processing. We target both 10-series and 30-series GPUs, demonstrating the portability of our approach.

Assembly begins with register allocation, allowing us to assign virtual registers in PTX to real registers in SASS. The number of registers is determined by the device, although remains consistent across the target platforms. For each intermediate function, we then map to the supported instruction set using optimized patterns. Most operations map 1-to-1 or decompose to an efficient hardware-dependent sequence.

As JIT-capability is a primary concern for our design, optimization is kept to a minimum and focuses on improving performance with low overhead. We thus focus on two improvements, instruction scheduling and peephole transformations. We assume that the generated code is largely optimal, getting the bulk of the performance from efficient parallel algorithms. This avoids needless costs from intensive fixed-point analyses and transformations typical to most optimizing compilers.

The last phase of assembly generates a binary ELF file compatible with the CUDA runtime, completing the compilation pipeline. SASS instructions are translated to binary using our framework, and augmented with NVIDIA-specific properties. We therefore remain independent of the proprietary pipeline while maintaining compatibility. The assembler is described in detail in Chapter 5.

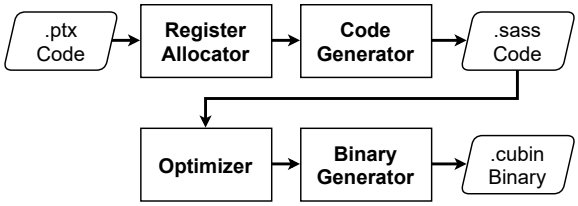


Figure 3.3 – rNdN backend assembler architecture.

3.3 Runtime

Once the query has been assembled to an executable form, it is passed to the runtime shown in Figure 3.4 along with the CPU control-code. The interpreter coordinates execution of the CPU code and dispatches kernels to the GPU for parallel sections. As the CPU is reserved for basic data manipulation functions, the interpretive overhead is negligible compared to the overall query. Alongside the interpreter, we provide an SQL library providing complex functions like sort, group, join and like. Each is implemented for the GPU using internal control-flow and data structures that are not efficiently expressible in HorseIR.

We build on-top of the NVIDIA CUDA runtime library and provide a unified interface for accessing and initializing the GPU. Expensive operations like data-caching, initialization, allocation and deallocation are optimized with versions suitable for query processing. An external library from NVIDIA provides complex math functions, compiled and optimized using LLVM.

Lastly, the data registry serves as the CPU-bound storage for the database, storing data in-memory and in columnar format. Data is automatically transferred and kept consistent between the two devices as the query is executed by the interpreter. The transfer cost, a significant bottleneck in GPU applications, is thus kept to a minimum. We discuss the runtime in Chapter 6.

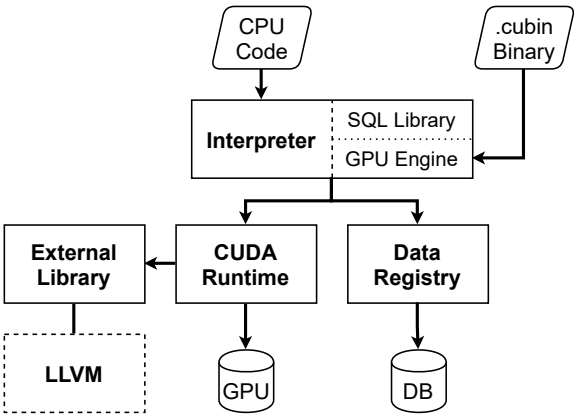


Figure 3.4 – rNdN runtime architecture.

Chapter 4
Frontend and Compiler

In this chapter we present the frontend and stage 1 compiler pipeline, beginning with the input HorseIR query program and describing the steps required to outline and generate efficient GPU code. We start with the frontend in Section 4.1, briefly detailing the parsing and semantics phases. In Section 4.2, we continue with the analysis framework and outliner, presenting our approach to automatically determining parallel sections (kernels) and CPU control code. Lastly, we describe the code generation strategy for built-in and library functions, thread layouts, and targeted optimizations in Section 4.3 before summarizing our approach in Section 4.4. The resulting PTX and CPU code are passed onto the Assembler and Runtime, described in Chapter 5 and Chapter 6 respectively.

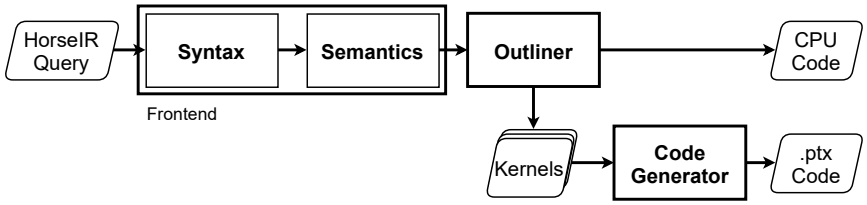


Figure 4.1 – rNdN frontend and compiler architecture.

4.1 Frontend

HorseIR is a statically-typed language that may either be compiled or executed in an interpreter [38]. Programs are defined in a `default` module comprising a collection of imports and user-defined functions. A `main` function serves as the entry point and kernels represent parallel sections as shown in Figure 4.2. Provided by the system, a `Builtin` module provides array-based and database functions, while an additional `GPU` module implements parallel algorithms requiring non-HorseIR control-flow or structure allocations. Query programs call a sequence of built-in and kernel functions before returning tabular data. Although not pictured, HorseIR has structured control-flow (loops and conditional statements) used in generic computation beyond SQL.

```
module default {
    import Builtin.*;                              // Array/database functions
    import GPU.*;                                  // GPU library functions
    def main() : table {                           // Query entry point, returns table
        var t0:i32     = [...];                    // Load input data
        var t1:i32     = @main_1(t0);              // GPU kernel call
        var t3:i64     = @GPU.order_lib(...);      // SQL `ORDER BY' library call
        var t4:list<?> = @list(t3);                // Collect columns for output table
        var t5:table   = @table(`order:sym, t4); // Form output table
        return t5;
    }
    kernel main_1(t0:i32) : i32 {                  // GPU kernel definition
        var t1:i32 = @plus(t0, 1:i32);             // Array built-in function
        return t1;
    }
}
module Builtin {                                   // Array and database functions
    def plus(t0:?, t1:?) : ? __BUILTIN__           // Element-wise binary `+'
    def load_table(t1:sym) : table __BUILTIN__     // SQL table loading
    [...]
}
module GPU {                                         // GPU library functions
    def group_lib(...) : dict<i64, i64> __BUILTIN__  // SQL group
    def order_lib(...) : i64 __BUILTIN__             // SQL sort
    [...]
}
```

Figure 4.2 – HorseIR syntax example, including built-in modules and kernel functions.

We implement a scanner+parser module using `flex` [2] and `bison` [1], parsing the input HorseIR query into an abstract syntax tree (AST). We then enforce semantic requirements with a symbol table and type checker as shown in Figure 4.3. The 3-pass symbol table resolves modules, imports, and functions calls to either user-defined functions and kernels, or library functions imported into the module. This is followed by type checking, which ensures type-correctness and infers wildcard types ("`?`") using built-in function rules. Type inference is required for static GPU code generation as the target language has no inference capability, and the input queries translated from SQL do not provide types for all variables. Lastly, we ensure variable declarations are defined on all paths before use with a definitely assigned analysis.

```
module default {
    import Builtin.*;
    import GPU.*;
    def main() : table {
        var t0:i32 = [...];
        var t1:str = [...];
        var t2:? = @Builtin.plus(t0, 1:i32); // Infer t2:i32; resolve @Builtin.plus
        var t3:? = @like_lib(t2, "%C":str); // Infer t3:bool; resolve @GPU.like_lib
        [...]
    }
}
```

Figure 4.3 – HorseIR semantics example, inferring wildcard types and resolving function calls/imports. Note that function calls can use the full path to resolve conflicts.

As an array-based language with SQL-specific functionality, HorseIR is well-suited for use in GPU databases. In particular, as array-based languages naturally expose data-dependencies and functionality is limited through the use of built-in functions, extracting parallelism is trivial compared to other imperative languages and the code generation strategies are simplified. Additionally, by directly supporting SQL functionality and types, traditional compiler techniques may be applied where necessary. As a compiler-first approach, this is especially relevant for high performance.

4.2 Outliner

SQL queries are translated to a single HorseIR function, mixing both sequential and parallel operations sourced from array-based languages and SQL databases. Before code generation, we must therefore identify sections of parallel code and outline them into efficient GPU kernels. The surrounding CPU-based control code can be executed in an interpreter without significant overhead as the primary computation is offloaded to the accelerator. Central to efficient outlining are 3 key ideas:

1. Minimize data access times, keeping data in fast memory;

2. Minimize redundant data accesses, loading data once; and

3. Maximize parallelism, keeping threads active.

We therefore define kernels as pipelines of *compatible* operations that are either data-dependent or share input data. This keeps intermediate data in fast registers memory, reduces reads and writes to slow device memory, and minimizes inactive threads that reduce parallelism. Compatibility, or the decision to combine operations into a single kernel, is thus essential to efficient execution and considers program properties, parallelism potential, and device capabilities. The full outlining pipeline has 5 main steps shown in Figure 4.4 and described in subsequent sections:

1. Data-dependency analysis and representation; (Section 4.2.2)

2. Shape analysis, recovering variable layouts; (Section 4.2.3)

3. Geometry analysis, mapping from shapes to threads; (Section 4.2.4)

4. Compatibility analysis, forming kernels; (Section 4.2.5)

5. Builder, generating the outlined form; (Section 4.2.6)

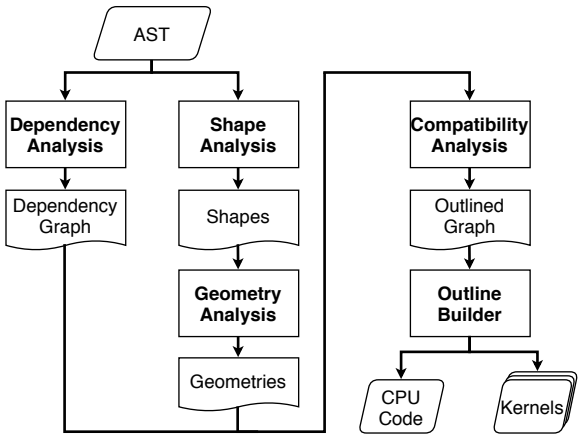


Figure 4.4 – GPU outliner pipeline for extracting kernels from input HorseIR queries. (© 2021 IEEE [125])

4.2.1 Framework

As part of our compiler, we provide a complete HorseIR data-flow analysis framework for analyzing queries and determining important program properties. Inspired by McSAF [52], the data-flow analyses traverse the structured control-flow of the program, exploiting loops and conditional statements provided by the IR to simplify analysis. Loops are conservatively analyzed using a fixed-point, ensuring computed properties are valid regardless of the iteration. Despite our supported SQL queries having no explicit control-flow, this generalizes our approach for future research on scientific computation. In addition, we provide a simple statement analysis that collects program properties independent of execution. Shown in Figure 4.5 is a simple live-variables data-flow analysis that determines, at each statement, which variables may be used at some point in the future. Note that as no control-flow is present, only a single iteration of the fixed-point analysis is represented. Recall that basic HorseIR types (e.g. `i32`) are *vectors*, with scalars values considered a vector of size 1.

As program analysis is the most expensive part of compilation, we spend considerable effort optimizing the data representation. In particular, we eliminate excess copying and allocation by storing objects (program properties) using pointers, allowing reuse between statements. Additionally, we reduce the data per-statement by half, storing either the `In` or `Out` sets depending on the use case. Lastly, sets

```
def main() : i32 {                      // Live variables
    var t1:i32 = [...];                 // In={}         Out={t1}
    var t2:i32 = @abs(t1);              // In={t1}       Out={t1, t2}
    var t3:i32 = @plus(t1, t2);         // In={t1, t2}   Out={t3}
    var t4:i32 = @mul(t3, 2:i32);       // In={t3}       Out={t4}
    return t4;                          // In={t4}       Out={}
}
```

Figure 4.5 – Example live-variables analysis for a HorseIR program without control-flow. At each point, we collect the set of variables that may be accessed in the future.

and maps provided by the standard library are replaced by optimized third-party implementations as they represent a significant cost of the overall pipeline [133].

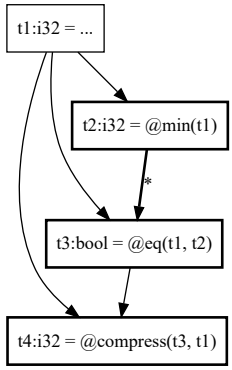
4.2.2 Program Representation

Compatibility analysis creates kernels by pipelining data-dependent operations, minimizing intermediate materialization and redundant accesses. We therefore begin by constructing an augmented data-dependence graph, capturing potential compatibility from dependencies and encoding parallelism metadata. A fixed-point data-flow analysis collects sets of statements that read and write each variable, and we subsequently form a data-dependence graph in which nodes are statements and edges are dependencies. Dependencies include write-read (true) dependencies, read-write (anti) dependencies, and write-write (output) dependencies. The result is combined with Shape and Geometry Analyses described in Sections 4.2.3 and 4.2.4 to outline kernels that also maximize parallelism. An example program and associated data-dependence graph is shown in Figure 4.6 and metadata subsequently described.

We augment the data-dependency graph with parallelism metadata, encoding the target architecture and synchronization and simplifying further analyses. Each node is tagged as either CPU or GPU, and all synchronized edges marked using rules defined below. CPU functions are limited to quick data management and challenging string manipulation, offloading the core computation to the GPU. We also identify library functions that operate as independent black boxes in the outline, as their

```
def main() : table {
    var t1:i32 = [...]
    var t2:i32 = @min(t1);
    var t3:bool = @eq(t1, t2);
    var t4:i32 = @compress(t3, t1);
    [...]
}
```

(a) Input Horse program.



(b) Augmented data-dependence graph.

Figure 4.6 – Example HorseIR program and augmented data-dependency graph. Bold nodes execute on the GPU; edges tagged '*' are synchronized. (© 2021 IEEE [125])

control flow or structure allocations preclude easy pipelining. A subset of function metadata is shown in Table 4.1, covering common and special cases.

Edges are synchronized in two cases: (1) write synchronization of the parent instruction; or (2) read synchronization of the child instruction. Write synchronization is required when the result scatters data globally across threads, as is the case with indexed writing and reductions. Read synchronization is conversely required for instructions that gather data across threads, as with indexed reads. Note that synchronization is variable dependent, as each variable has different read and write properties. For a pair of data-dependent instructions, the edge must be synchronized if read or write synchronization is required for any dependency.

We exploit two exceptions present in SQL queries to improve performance by eliminating intermediate synchronization. Firstly, boolean masking (`@compress`) performs an indexed write, typically requiring synchronization and writing the output to device memory. In our approach, we selectively disable GPU threads according to the mask and delay compression until absolutely necessary (e.g. read synchronization). If the result is aggregated (common for SQL), synchronization and writing are eliminated entirely. Secondly, Chen et al. addressed a common occurrence for reductions on lists which produces a single value per cell [39]. If the subsequent operation converts each cell to an element in a vector (`@raze`), we can generate the vector in place.

Table 4.1 – Parallelism metadata for a subset of HorseIR functions.

Function	Target	Synchronization		Description
		Read	Write	
`@eq(a, b)`	GPU	✗	✗	Element-wise comparison, no synchronization required.
`@compress(a, b)`	GPU	✗	✗	Boolean compression, no synchronization required (optimized by disabling threads).
`@index(a, b)`	GPU	a:✗; b:✓	✗	Indexed-read (gather), requires read synchronized data.
`@index_a(a, b, c)`	GPU	a:✓; b:✗; c:✗	✓	Indexed write (scatter), requires read synchronized data, write synchronized result.
`@min(a)`	GPU	✗	✓-reduction	Reduction, write synchronized result.
`@raze(a)`	GPU	✗	✓-raze	Convert list-of-vectors to a single vector, write synchronized result.
`@load_table(a)`	CPU	-	-	Load SQL table from datastore.

4.2.3 Shape Analysis

An efficient GPU kernel maximizes parallelism, avoiding excess inactive threads. We therefore statically analyze the data layout and size of each variable, determining its abstract shape. The result is subsequently used in Geometry analysis to compute efficient thread layouts of each parallel operation and determine compatibility.

For each HorseIR type we define an abstract *shape* parameterized by one or more *sizes*. Shape and size definitions are shown in Table 4.2 and Table 4.3 respectively. *Vector* data represents a contiguous array, allocated with fixed length. *Lists* represent collections of vectors, one per cell, with each cell potentially differing in size. As a convenience, if only one vector is provided, it applies to all cells. For databases, we support *enumeration* as foreign keys, *dictionary* for grouping, and *table* for input and output relations. Enumerations are parameterized both by the size of the foreign key, as well as the size of the remote column. Grouping collects elements into bins according to a common value. The result is represented as a dictionary, with each value associated to a vector. Lastly, table data is parameterized by the number of rows and columns. In generalized HorseIR, compound types list, enumeration and dictionary may be parameterized by other shapes not used in SQL.

Table 4.2 – Shape abstractions for SQL queries. (Adapted, © 2021 IEEE [125])

Shape	**Description**
Vector(N)	Array of size N
List(N, [Vector])	Collection of N Vectors, one per cell
Enumeration(N, M)	Foreign key of size N, pointing to Vector of size M
Dictionary(N, List(N, [Vector]))	Dictionary of N key-value pairs
Table(N, M)	Tabular data of size N x M

Kernel outlining occurs statically, before the program is executed. Size representations must therefore be powerful enough to identify identical allocations used for compatibility. We define 4 size kinds, capturing common cases shown in Table 4.3. Vector literals can be represented exactly at compile time with *constant*, while data loaded from input relations is represented as a *symbol*. We can guarantee symbolic correctness as input tables are immutable throughout execution. *Compression* is expressed recursively, associating the initial size with the boolean mask. We can thus recover the initial size, and identify compressions which share behaviour. Lastly, *dynamic* sizes capture function outputs that depend on exact content such as `@unique`. We associate each dynamic shape with a unique identifier used to determine equality.

Table 4.3 – Size abstractions for SQL queries. (© 2021 IEEE [125])

Size	**Description**
Constant(k)	Compile-time constant
Symbol(s)	Symbolic input size
Compressed(Size, Mask)	Boolean masked size
Dynamic	Runtime dependent size

Note that compressed sizes include the boolean mask in their representation, identifying variables compressed with the same value. This is required, as compressed shapes with same size but different masks require distinct thread layouts. Before Shape analysis, we thus associate each variable with a unique *data object* using an auxiliary analysis. In the case of divergent control-flow, we associate the merged program variables with new unique objects. An example HorseIR program and its Shape analysis are shown in Figure 4.7. Boolean masks are indicated by the variable name.

```
def main() : table {
    var t1:i32 = [...] // `amount column
    var t2:i32 = @min(t1);
    var t3:bool = @eq(t1, t2);
    var t4:i32 = @compress(t3, t1);
    var t5:list<i32> = @list(t4);
    var t6:table = @table(`out, t5);
    return t5;
}
```

(a) Input Horse program. Variable `t1` loads data from the `‘amount` column.

Variable	Shape
t1	Vector(‘amount)
t2	Vector(1)
t3	Vector(‘amount)
t4	Vector(‘amount[t3])
t5	List(1, Vector(‘amount[t3]))
t6	Table(1, ‘amount[t3])

(b) Shape analysis output for each variable. Note `‘amount[t3]` is shorthand for Compressed(`‘amount`, DataObject(`t3`))

Figure 4.7 – Example shape analysis output for the example HorseIR program.

Definition (“Laurie’s 6 steps” [98, 225])

Shape analysis is a fixed-point static analysis, approximating the data layout and size of each variable at runtime. Following is the formal analysis definition.

Step 1: Approximation

We collect sets of tuples (variable, shape) mapping each variable to its allocated shape and size at each program point. The shape and size abstractions are defined in Tables 4.2 and 4.3 respectively.

Step 2: Definition

Let `v` be a variable defined at program point `d`. At program point `p`, variable `v` has shape `s` if on all paths from `d` to `p` variable `v` has shape `s`, and variable `v` is not redefined between `d` and `p`.

Step 3: Direction

Forward analysis, collecting shapes about the execution up to program point `p`.

Step 4: Merge Operation

We merge control-flow paths by a modified set union, recursively merging shapes for each variable into a single representation. We define the merge operation for each shape in Figure 4.8. Intuitively, we return the most specific result

possible, maintaining the shape kind and size if equal. For compound types list and dictionary, we pair-wise merge element shapes. Sizes must either be equal, or a new dynamic size is created. For compression, only the base sizes must be equal - differing masks create a new unique object.

```
def MergeShape(shape1, shape2):
    if shape1.kind != shape2.kind:
        return Wildcard()
    switch shape1.kind:
        case Vector:
            size = MergeSize(shape1.size, shape2.size)
            return Vector(size)
        case List:
            size = MergeSize(shape1.size, shape2.size)
            if shape1.size == shape2.size:
                elements = MergeShapes(shape1.shapes, shape2.shapes)
                return List(size, elements)
            else:
                return List(size, [VectorShape(Dynamic)])
        case Enumeration:
            size = MergeSize(shape1.size, shape2.size)
            foreign = MergeSize(shape1.foreign, shape2.foreign)
            return Enumeration(size, foreign)
        case Dictionary:
            size = MergeSize(shape1.size, shape2.size)
            if shape1.size == shape2.size:
                elements = MergeShapes(shape1.shapes, shape2.shapes)
                return Dictionary(size, List(shape1.size, elements))
            else:
                return Dictionary(size, List(size, [VectorShape(Dynamic)]))
        case Table:
            rows = MergeSize(shape1.rows, shape2.rows)
            cols = MergeSize(shape1.cols, shape2.cols)
            return Table(rows, cols)

def MergeSize(size1, size2):
    if size1 == size2: # For compressed sizes, includes substructures
        return size1
    if size1.kind == Compressed and size2.kind == Compressed:
        if size1.size == size2.size: # Sizes equal, masks different
            return Compressed(size1.size, DynamicObject)
    return Dynamic()
```

Figure 4.8 – Merge rules for shape and size abstraction.

Step 5: Starting Approximation

For the function entry point, we add a dynamically-sized shape for each parameter (Figure 4.9), or the function call shapes in the case of interprocedural analysis. Other basic blocks are initialized with the empty set, computing the least fixed-point and providing useful shape information for subsequent steps.

- in(Entry) = {(v, `DynamicShape(v.type)`) | v ∈ parameters}
- in(Entry) = {(v, `CallShape(v)`) | v ∈ parameters}, interprocedural
- in(B_i) = {}, for other basic blocks

```
def DynamicShape(type):
    switch type:
        case Vector:
            return Vector(Dynamic)
        case List:
            // Dynamic1 may be replaced by a constant if defined in the type
            return List(Dynamic1, [Vector(Dynamic2)])
        case Enumeration:
            return Enumeration(Dynamic1, Dynamic2)
        case Dictionary:
            return Dictionary(Dynamic1, List(Dynamic1, [Vector(Dynamic2)]))
        case Table:
            return Table(Dynamic1, Dynamic2)
```

Figure 4.9 – Dynamic shape initialization for function parameters.

Step 6: Flow Equations

HorseIR consists primarily of assign statements, generating a new tuple for each assigned variable. We use the following flow equation, where the gen set for statement S_i is given by the specific built-in or user-defined function and the kill set removes previous tuples of each defined variable.

$$\text{out}(S_i) = (\text{in}(S_i) \setminus \text{kill}(S_i)) \cup \text{gen}(S_i)$$

We define the shape rules for each built-in function and interprocedural analysis for user-defined functions. The rules for unary and binary element-wise vector functions are shown below as well as the extension to lists.

Table 4.4 – Shape rules for binary element-wise vector functions. (S, C, and D are short for Symbolic, Compressed and Dynamic; only the Vector size is shown)

		Shape b				
		1	**N**	$\mathbf{S}_b$	$\mathbf{C}_b$	$\mathbf{D}_b$
Shape c	**1**	1	N	S_b	C_b	D_b
	M	M	if M == N: M else: Error	Dynamic	Dynamic	Dynamic
	$\mathbf{S}_c$	S_c	Dynamic	if S_b == S_c: S_b else: Dynamic	Dynamic	Dynamic
	$\mathbf{C}_c$	C_c	Dynamic	Dynamic	if C_b == C_c: C_b else: Dynamic	Dynamic
	$\mathbf{D}_c$	D_c	Dynamic	Dynamic	Dynamic	if D_b == D_c: D_b else: Dynamic

Unary: `a = @function(b)`

$$\text{gen}(S_i) = \{(\texttt{a, b.shape})\}$$

Unary functions operate on each vector element individually, propagating the input shape and producing a same-size Vector.

Binary: `a = @function(b, c)`

$$\text{gen}(S_i) = \{(\texttt{a, Table(b.shape, c.shape)})\}$$

Binary functions are more complex, as they may either broadcast a scalar value, or perform an element-wise computation. Given constant values, we can compute the exact resulting shape. Otherwise, we require equality before falling back to dynamic. Full shape rules are defined in Table 4.4.

Lists: `a = @each(@function, b)`

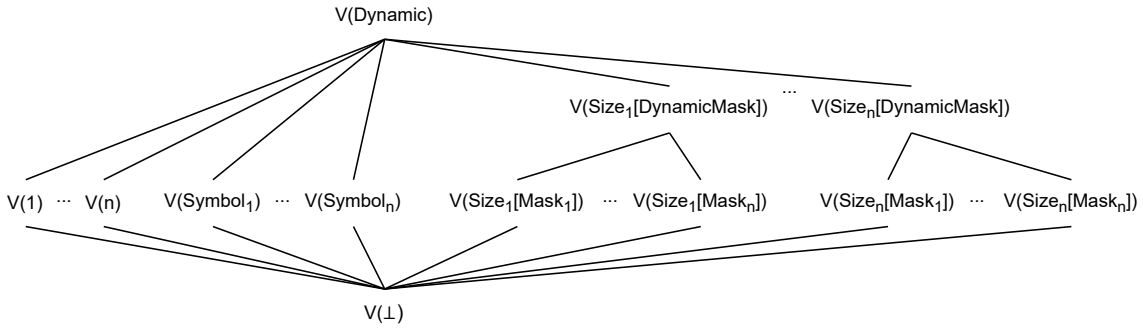
$$\text{gen}(S_i) = \{(\texttt{a, List(b.size, [@function(e) for e in b.shapes])})\}$$

List functions operate on each cell in isolation (or pair-wise for binary functions taking two lists). We therefore recursively analyze the element shapes, applying the function to each cell separately. The resulting list shape has the same size, and one element shape per cell.

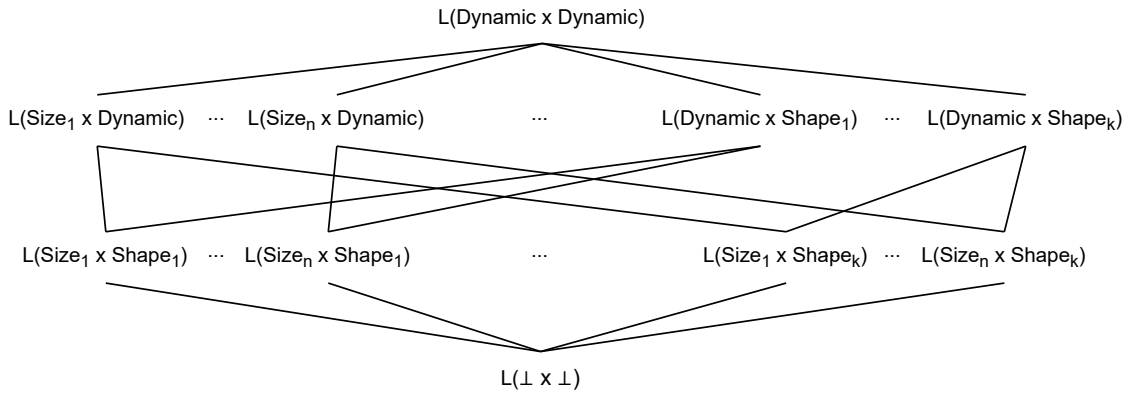
To ensure termination, we define our flow and merge functions to be monotonic on a finite domain that contains a greatest element and a least element.

- **Least element** ($\bot$): Uninitialized shape/size
- **Greatest element** ($\top$): Wildcard shape/dynamic size

We conceptually divide the domain into separate sections, one per shape, each joined at the least and greatest elements in the overall domain. The sections for vector and list shapes are shown in Figure 4.10 and explored below.



(a) Vector shape.



(b) List shape.

Figure 4.10 – Shape analysis domains and partial orderings for list and vector shapes. The overall domain connects the least elements to a generic uninitialized shape, and the greatest elements to a generic wildcard (not pictured).

Vector shapes are parameterized by one of 3 sizes: constant, symbolic, and compressed. The ordering is mostly flat, as vectors of different shapes have no relation. We do, however, allow for compressed vectors of the same size but different boolean masks as their relation is based on runtime data.

List shapes are parameterized by the Cartesian product of the size (the number of cells), and the cell shape. We assume only a single cell shape for simplicity, but the idea can be extended to uneven cells. The ordering allows for dynamism in both elements independently as it gives more precise analysis results.

4.2.4 Geometry Analysis

Given the shapes of each program variable, we compute the *geometry* of each statement, abstracting its thread layout and determining compatible operations. As we parallelize only the core computation, geometry is limited to vector and list operations. All other operations are CPU-based and have no associated geometry. An example program and its associated geometries are shown in Figure 4.11, operating on vector data. We describe the mapping from geometry to thread layouts used for code generation and execution in Section 4.3.2.

```
1 def main() : table {
2     var t1:i32 = [...] // `amount column
3     var t2:i32 = @min(t1);
4     var t3:bool = @eq(t1, t2);
5     var t4:i32 = @compress(t3, t1);
6     var t5:list<i32> = @list(t4);
7     var t6:table = @table(`out, t5);
8     return t5;
9 }
```

(a) Input Horse program. Variable `t1` loads data from the `‘amount` column.

Statement	Geometry
2	CPU
3	Vector(‘amount)
4	Vector(‘amount)
5	Vector(‘amount)
6	CPU
7	CPU

(b) Geometry analysis output for each statement.

Figure 4.11 – Example geometry analysis output for the example HorseIR program.

Geometry is a statement-based analysis, traversing the program and determining the geometry for each statement in isolation. Using the result from shape analysis, we

compute the geometry using a set of predefined rules, one for each built-in function. These rules map the parameter and return shapes to an abstract thread layout suitable for GPU processing. User-defined functions are omitted in our approach, focusing on self-contained kernels, but can be supported with interprocedural analysis. Shown below are the geometry rules for select classes of built-in functions.

Unary: `a = @function(b)` **Geometry = Shape(a)**

Unary functions propagate the input shape, performing an element-wise computation. The operating geometry is thus the shape of the output vector.

Binary: `a = @function(b, c)` **Geometry = Shape(a)**

Binary functions either broadcast from a scalar value onto a vector, or perform an element-wise computation. We can simplify the analysis by taking the output shape of the function, as it corresponds to the amount of work performed.

Compression: `a = @compress(b, c)` **Geometry = Shape(b)**

Compression masks an input vector according to a same-size boolean array. The operating geometry is thus the shape of the input vector.

Reduction: `a = @function(b)` **Geometry = Shape(b)**

Reductions produce a single value for a vector shape. The geometry thus corresponds to the shape of the input vector.

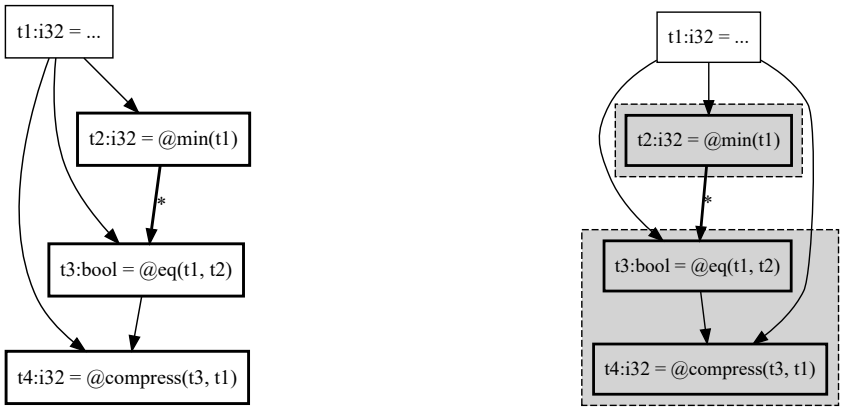
Lists: `a = @each(@function, b)` **Geometry = Analyze(@function, b)**

For list operations, we use a recursive approach to computing geometry, analyzing the vector geometry of each cell using the nested function. We then form the list geometry which parallelizes cells and their contents. The list size is propagated from the input as the number of cells does not change.

4.2.5 Compatibility Analysis

Compatibility analysis extracts parallelism from a HorseIR query program, given the augmented data-dependence graph and geometry analysis. Each parallel region or kernel is defined such that parallelism is maximized, while slow data accesses

and intermediate materializations are minimized. Operating on the augmented data-dependence graph, the analysis merges *compatible* operations in a greedy manner. Compatible operations are those that may benefit from being merged into a single kernel, minimizing data access costs while maintaining parallelism as detailed in our 3 step algorithm defined below. The resulting kernels satisfy data-dependency and synchronization requirements, and efficiently exploit parallelism. We continue the previous example from geometry analysis (Figure 4.11), showing outlined regions in grey in Figure 4.12. Each region represents a parallel fragment that can be translated to a single GPU kernel during code generation. Only the main computation is shown, as CPU-resident operations are not formed into kernels.



(a) Augmented data-dependence graph.

(b) Outlined data-dependence graph.

Figure 4.12 – HorseIR program outline, showing outlined regions from compatibility analysis in grey. Synchronized edges are tagged '*', and GPU-capable nodes are bold. CPU-only nodes (`t5`, `t6`) forming the output table are omitted. ([125])

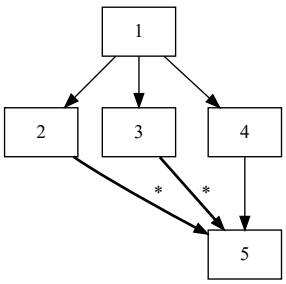
We use a 3 step greedy algorithm to identify kernels, illustrated by example for a simple program fragment in Figure 4.13. By outlining kernels directly in the high-level HorseIR program, we simplify the design and implementation of subsequent compiler phases. Each step focuses on a specific merging case that optimizes data accesses while maintaining high degrees of parallelism.

1. Identify data-dependent, unsynchronized operations with identical geometry;

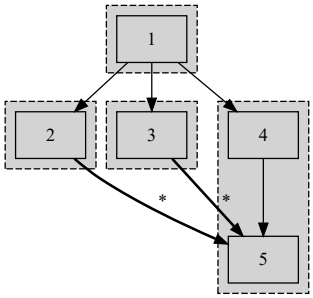
2. Merge kernels with common input data; and

3. Merge parent-child kernels and optimizing compressed geometry.

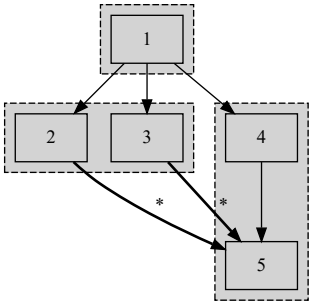
Care must be taken in outlining the graph to avoid data-dependent cycles between kernels as HorseIR programs are acyclic other than well-structured loops (cycles are only implied in the representation). We must also ensure that input and output synchronization requirements are satisfied. Unsupported cycles may be introduced if the input and output of an operation are optimistically merged into a single kernel, but the operation itself cannot be included due to synchronization. Each step is described in detail below, along with its associated algorithm.



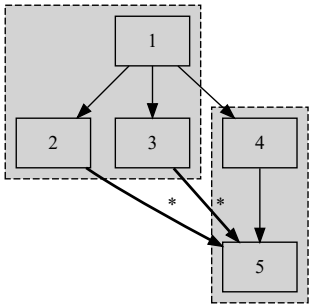
(a) Input data-dependence graph. Synchronized edges tagged by '*'.



(b) Step 1: Identify data-dependent operations with no synchronization.



(c) Step 2: Merging sibling kernels (2)-(3) with common input data.



(d) Step 3: Merge parent kernel (1) with the *best* legal successor (2-3). Note that merging (1) with (4-5) would introduce a kernel cycle.

Figure 4.13 – Compatibility analysis steps breakdown. All nodes have identical geometry and are GPU-capable. (© 2021 IEEE [125])

Step 1

The first step of outlining merges data-dependent GPU operations with identical geometries, focusing on high-payoff pipelines. We traverse the graph reverse-topologically, tagging GPU-capable operations with their respective kernel. For each statement, we may either extend an existing kernel, or form a new parallel region. Extension is possible if the following are all true, otherwise we start a new kernel:

1. All successors belong to the same kernel;

2. The kernel geometry is identical to the statement; and

3. All successor edges are unsynchronized.

Requiring all successors belong to the same kernel avoids creating pairs of interdependent kernels without cycle detection. Applied to the example program in Figure 4.13b, only nodes 4 and 5 may be merged. Edges 2-3 and 3-5 are synchronized, and node 1 has multiple successor kernels. The complete algorithm is shown in Figure 4.14.

Step 2

Given the kernels identified the step 1, we optimize the result in 2 integrated phases, representing merged kernels as super nodes in an updated graph. Traversing forward-topologically, we merge sibling kernels for each node, provided that they are not connected by a path. If a path exists but separate kernels were formed in step 1, then there must exist a synchronized data-dependency that prevents merging. We additionally require that the geometries are identical, maximizing parallelism. This steps reduces redundant input data loads, while satisfying synchronization. Shown in Figure 4.13c, independent sibling kernels 2 and 3 may merged as they both load input data from node 1 and have identical geometry. No merging is possible with super node 4-5 due to synchronization. The first optimization step is shown in Figure 4.15.

```
kernelMap = {}

def Outline(graph, geometries):
    // Build kernels bottom-up, greedily identifying pipelines
    for statement in ReverseTopologicalOrder(graph):
        if graph.IsGPU(statement):
            kernel = SelectKernel(statement, geometries[statement], graph)
            if kernel == None:
                // Start a new kernel if none found
                kernel = Kernel(geometries[statement])
            kernelMap[statement] = kernel
        else:
            kernelMap[statement] = None

def SelectKernel(statement, geometry, graph):
    kernel = None
    for successor in graph.Successors(statement):
        // Synchronized edges create a new kernel
        if graph.IsSynchronized(statement, successor):
            return None

        // Ensure that all successors are in the same kernel
        if kernel == None:
            kernel = kernelMap[successor]
        else if kernel != kernelMap[successor]:
            return None

    // Kernel geometry must be equal to the statement
    if !IsEqual(kernel.geometry, geometry):
        return None
    return kernel
```

Figure 4.14 – Step 1: Identify data-dependent operations with identical geometry.

```
def MergeLoads(node):
    // Iterative merging of successor nodes with equal geometry
    for successor1, successor2 in graph.SuccessorPairs(node):
        // Ensure no path between the kernels (required by synchronization)
        if graph.ContainsPath(successor1, successor2):
            continue

    // Merge sibling kernels if geometry identical
    if IsEqual(successor1.geometry, successor2.geometry):
        graph.MergeNodes(successor1, successor2)
```

Figure 4.15 – Step 2: Merge kernels with input dependencies.

Step 3

Once the successor kernels are merged in the previous step, we then merge the current node with the *best* compatible successor that does not introduce a cycle. The best successor is that which shares the most data-dependencies, reducing intermediate materialization. We also optimize compression, allowing longer pipelines which incrementally mask the active geometry. This is especially useful for queries which aggregate a subset of rows, avoiding the write synchronization required for compressed data entirely. For this step only, we thus relax compatibility to include compressed geometry. In particular, we allow the best successor geometry to be a *subset* of the parent node. The resulting kernel uses the full, unmasked geometry for execution. Completing the example in Figure 4.13d, we merge the parent kernel with the 2-3 super node as it is the only choice that does not introduce a cycle between kernels. The alternative, merging the parent kernel with the 4-5 super node would no longer be acyclic. Note that compression is not considered as all geometries in the example are identical. The algorithm for selecting the successor kernel is shown in Figure 4.16. Note that as a greedy algorithm, merging some kernels may in fact hurt performance as the gains from eliminating intermediate materialization may be more than offset by reductions in parallelism from compressed geometries or register pressure.

```
def MergeCompress(node):
    bestSuccessor = None
    bestSize = 0

    for successor in graph.GetSuccessors(node): // Successor loop
        // Synchronized edges preclude merging
        if graph.IsSynchronized(node, successor):
            continue

        // Ensure no edges from any other successor (creates a cycle)
        for successor2 in graph.GetSuccessors(node):
            if graph.ContainsDirectedPath(successor2, successor):
                continue // Successor loop

        // Geometry must be compatible, allowing compressed successor size
        if !IsCompatible(node.geometry, successor.geometry):
            continue

        // Maximize the dependency count, reducing materialization
        size = graph.GetDependencyCount(node, successor)
        if size > bestSize:
            bestSuccessor = successor
            bestSize = size

    // Merge the current node with its best successor
    if bestSuccessor != None:
        graph.MergeNodes(node, bestSuccessor)
```

Figure 4.16 – Step 3: Merge parent-successor kernels with compressed geometries.

Control-Flow

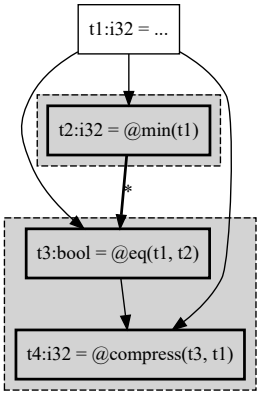
HorseIR supports structured control-flow through loops and if-else conditional statements. While not present in SQL queries, compatibility analysis extends to such structures in a recursive manner. The outliner first traverses the body (loops) or branches (if-else) recursively before proceeding as normal, considering each structure a super node in the graph. We can therefore generate kernels with limited control-flow, although the compilation cost will be higher due to the required fixed-point shape analysis and recursive traversal.

Loops: Loops in HorseIR come in two variations `while` and `repeat`. Both iteratively execute a sequence of statements shielded by a scalar condition, common to all threads. We can therefore translate any loop whose body statements have identical geometry and form a single kernel. Note that scalar operations are compatible with any kernel geometry, as they can be executed once per thread. We additionally require that the body has no synchronization, leaving such cases to the interpreter which can repeatedly call the loop body kernel. Loop carried dependencies are thus allowed as long as no synchronization is present – we are parallelizing the loop body and not the iterations.

If-Else: Conditional statements follow the traditional format, with scalar condition similar to loops. They may be translated to GPU code if each branch forms a single kernel, and have identical geometry. We require identical geometry for simplicity, even though divergence is not possible. Synchronization is allowed, provided that it ends the branch (otherwise multiple kernels are required).

4.2.6 Builder

The last step in the outliner translates the graph back to HorseIR, constructing new kernels, library calls and the CPU control-code. Statements are ordered using a forward-topological traversal, ensuring that variables are defined before use. The program is then run through the semantics phase again, ensuring correctness at low cost. The completed example program is shown in Figure 4.17.



(a) Outlined data-dependence graph.

```
def main() {
    var t1:i32 = [...]
    var t2:i32 = @main1(t1);
    var t4:i32 = @main2(t1, t2);
}
kernel main1(t1:i32):i32 {
    var t2:i32 = @min(t1);
    return t2;
}
kernel main2(t1:i32, t2:i32):i32 {
    var t3:bool = @eq(t1, t2);
    var t4:i32 = @compress(t3, t1);
    return t4;
}
```

(b) Outlined HorseIR program.

Figure 4.17 – HorseIR program translation from outline to code. (© 2021 IEEE [125])

Library functions require complex internal control-flow or specific structure allocations that are not present in HorseIR. They are also type-specific and are therefore generated at compile time for each query program, although caching is possible for frequent types. An outlined library call decomposes into the invocation, and one or more kernels for the core computation. Code generation implements the templates, and the runtime is responsible for execution. Shown in Figure 4.18 is a library call to the `@order` function. The library patterns are described in Section 4.3.4.

```
def main() {
    var t1:i32 = [...]
    var t2:i64 = @GPU.order_lib(@order_init, @order, t1);
}
kernel order_init(data:i32) : i64, i32 {
    var index:i64, data_out:i32 = @GPU.order_init(data, 0:bool);
    return index, data_out;
}
kernel order(index:i64, data:i32) {
    @GPU.order(index, data, 0:bool);
}
```

Figure 4.18 – Outlined library call to sorting function `@order`.

4.3 Code Generation

Our code generation strategy targets PTX, a low-level intermediate representation from NVIDIA designed for GPUs [170]. It follows a simplistic approach, mapping from high-level HorseIR functions to a parallel GPU-friendly language without extensive optimization. We describe the PTX language and our implementation framework in Section 4.3.1. Thread layouts and code generation templates are discussed in Sections 4.3.2 and 4.3.3, while library function implementations are covered in Section 4.3.4. Lastly, we briefly present limited optimizations in Section 4.3.5.

4.3.1 Target Language: PTX

PTX, or *Parallel Thread Execution*, is general-purpose intermediate representation for GPUs [170]. Designed to be platform independent, functionality is attached to *compute capabilities*, each representing a class of hardware. For portability reasons, we only use features supported by capability `sm61` (Pascal) and `sm86` (Ampere) in PTX version 7.4. PTX code is assembled by the driver to a device specific instruction set (SASS), and the resulting binary is executed by the CUDA runtime. By generating PTX instead of CUDA, we bypass the first compilation phase and reduce the compile time. An example PTX program that increments an array is shown in Figure 4.19.

PTX programs are split into modules, each defining a set of module-scope variables and functions. They also specify the version, target (compute capability) and the address size for pointers (64-bit for our architectures). Linking directives may be used to extend symbol visibility to program-scope and import external declarations from other modules. Kernels are tagged as entry points accessible by the runtime, while other functions are only accessible from GPU code. Functions contains a list of parameters and a statements block.

PTX is a statically-typed language, comprising signed and unsigned integers, floating points and boolean predicates. For type-agnostic operations like shifting, untyped "bit" variables are also supported. Variables are defined with their type and *state space*, indicating their storage in the GPU memory hierarchy. Spaces include global

```
.version 7.4
.target sm_61
.address_size 64

.visible .entry add(.param .u64 input, .param .u64 output) {
    .reg .b32       %r<7>;
    .reg .b64       %rd<8>;
    // Load input and output addresses from parameters
    ld.param.u64    %rd1, [input];
    ld.param.u64    %rd2, [output];
    cvta.to.global.u64      %rd3, %rd2;
    cvta.to.global.u64      %rd4, %rd1;
    // Compute the global thread index using the thread block and local indexes
    mov.u32         %r1, %ctaid.x;
    mov.u32         %r2, %ntid.x;
    mov.u32         %r3, %tid.x;
    mad.lo.s32      %r4, %r2, %r1, %r3;
    cvt.u64.u32     %rd5, %r4;
    // Compute the input address and load the value
    add.s64         %rd6, %rd4, %rd5;
    ld.global.u8    %r5, [%rd6];
    // Increment value by 1
    add.s32         %r6, %r5, 1;
    // Compute the output address and store the new value
    add.s64         %rd7, %rd3, %rd5;
    st.global.u8    [%rd7], %r6;
    ret;
}
```

Figure 4.19 – Example PTX program incrementing each value in an array.

(device memory), shared, registers, and an additional parameters space for passing constant values to kernels. Input and output buffers are passed by their address as an unsigned integer. Variable declarations may be parameterized by a size, defining multiple variables with a common prefix.

Instructions are likewise typed, and require that arguments match the specified type. Data conversions are explicit, with the exception of bit-typed instructions and variables. Instructions include arithmetic and comparison operations for computation, and movement and synchronization for accessing and ensuring consistency of data. For data operations like loading, the state space is also included.

We implement a typed PTX representation using C++ templates, enabling type-correctness in our code generation. Each instruction is templated, statically ensuring arguments match the required types and space. If incorrect arguments are used, an error is generated when building the code generator rather than at runtime. We also selective enable and disable instruction features (e.g. saturation) depending on the type, ensuring no improper flag can be set. In cases where the type rules are relaxed, explicit adapter classes allow converting between types. Shown in Figure 4.20 is a fragment of code generation for correct, incorrect, and adapted types.

```
// Typed arguments, including registers and immediate values
auto destination = new PTX::Register<PTX::UInt64Type>("%d");
auto sourceA = new PTX::Register<PTX::UInt64Type>("%s");
auto sourceB = new PTX::Value<PTX::UInt64Type>(100);

// Type correct add instruction, corresponding to "add.u64 %d, %s, 100"
auto add = new PTX::AddInstruction<PTX::UInt64Type>(
    destination, sourceA, sourceB
);

// Type incorrect move instruction, error when building code generator
//     .u32 instruction with .u64 operands
auto move = new PTX::MoveInstruction<PTX::UInt32Type>(destination, sourceA)

// Untyped 64-bit variable (.b64)
auto untypedA = new PTX::BitAdapter<PTX::UIntType, PTX::Bits::Bits64>(source);
```

Figure 4.20 – PTX framework example showing the type system.

4.3.2 Thread Layout

Each kernel has an associated geometry, abstractly representing the thread layout. Before generating parallel code, we must therefore define a mapping that will be used to partition data between threads. Each kernel loads data by a geometry-dependent indexing function before generating the body, as discussed in Section 4.3.3. We therefore maintain independence between the geometry and code generation patterns, simplifying the pipeline.

GPU device memory is accessed by transactions, loading or storing[1] an aligned data segment in a single operation [161]. *Coalescing* merges accesses to the same segment into a single transaction, optimizing bandwidth and improving performance. Efficient GPU code thus requires that threads belonging to the same warp access neighbouring data locations, canonically reduced to the idiom "consecutive threads access consecutive locations" by GPU programmers. We show a comparison of efficient and inefficient memory access patterns in Figure 4.21.

(a) Efficient memory access pattern.

(b) Inefficient memory access pattern.

Figure 4.21 – GPU device memory transactions (8-byte data, 32-byte transaction). Adapted from NVIDIA CUDA documentation [161].

Our approach supports vector and list geometries, with all other operations executed on the CPU. The corresponding layouts are designed to coalesce accesses, assuming data is not gathered/scattered. We illustrate the thread mappings in Figure 4.22 and describe them below. All parallelism is simplified to a single dimension.

Vector geometry: Vector geometry assigns a single thread per element, each operating on data corresponding to its global thread index.

List geometry: List geometry assigns a fixed sequence of threads to each cell, together computing the result. For simple and efficient mapping, each cell gets the same number of threads. Some threads may therefore be inactive, or operate on multiple elements in a loop.

[1] https://stackoverflow.com/questions/20186744/memory-coalescing-in-global-writes

(a) Vector geometry: 1 thread per data element. (b) List geometry: Fixed threads (4) per cell.

Figure 4.22 – Thread layout for GPU kernel geometries. (© 2021 IEEE [125])

4.3.3 Templates

Code generation translates each high-level HorseIR built-in function to a parallel GPU equivalent. Each template is simplistic, using minimal low-level optimization and focusing instead on high-payoff algorithm design. The generated code is therefore easier to understand and analyze, while performing well on real queries. We depend on the assembler phase for architecture specific optimization.

Type Support

We begin by mapping vector HorseIR types to those in PTX as shown in Table 4.5, with integer and floating point mapping trivially. Lists are decomposed into their cell types for computation. Predicate values are stored in `.pred` for comparison and branching, but must accessed as 8-bit integers in device memory. Strings and symbols are stored as 64-bit hashed values, also referred to as *dictionary encoding* [61]. This limits the kind of computation possible on the GPU, but is sufficient in practice. Lastly, date and time values are mapped to signed integers for epoch time.

Table 4.5 – Mapping from HorseIR types to PTX.

HorseIR	PTX
bool	.pred
char	.s8
i8	
i16	.s16
i32	.s32
i64	.s64
f32	.f32
f64	.f64

HorseIR	PTX
str	.u64
sym	
date	.s32
month	
minute	
second	
dt	.s64
time	

Functions and Geometry

We split code generation into two components: (1) control code, and (2) expressions. Control code includes loading parameters, and mapping from threads to indexes used to load and store data. This allows us to isolate geometry dependent code from the core computation in most instances, simplifying generation.

Each function contains input and return parameters as well as the exact runtime geometry. Parameters are passed as a pair of pointers in device memory, corresponding to the data and size buffers. Data sizes are used for bounds checking of loading and storing results, as the number of threads does not necessarily match the data. This may be due to device requirements (the number of threads in each block must be the same), or accessing data of unknown size. Note that although input buffer sizes are constant and could be passed directly to the function, we prefer to leave data GPU-resident wherever possible. We also include the runtime geometry as a parameter, as it defines the *active* threads. An example signature for a function with one input and one output parameter is shown in Figure 4.23, each described as a pair of pointers to device memory. Data allocations are further described in Chapter 6.

```
.visible .entry kernel(
    .param .u64 input_data,
    .param .u64 input_size,
    .param .u64 output_data,
    .param .u64 output_size,
    .param .u32 geometry
)
```

Figure 4.23 – Example PTX function signature for one input parameter and one return parameter. Vector geometry is passed as an unsigned integer.

Vector computation is simplistic, as each thread is responsible for a single unit of data. Code generation thus initializes parameters and loads the runtime geometry before generating the function body statement-by-statement. Data is loaded and stored depending on its shape, either as a vector, or broadcast from a single value as shown in Figure 4.24a. For loading and storing compressed data, we use a global

prefix sum based on StreamScan [241] to convert the boolean mask to an index, similar to the idea from Funke et al. in their GPU database system [68]. Described in other work as *chained-scan*, thread blocks are executed in order and the prefix sum *propagated* from one to the other using device memory [147]. The propagation and block ordering variables (32-bit each) are loaded and stored as packed data (64-bit) for memory consistency [147]. As thread block indexes do not always relate to the launch order, they must be "dynamically allocated" during execution [241, 76].

List computation is more complex, assigning a fixed number of threads per cell. The exact size is chosen by the runtime system described in Chapter 6, and depends on the data cell sizes. To handle cases where the data size exceeds the allocated threads, we iterate over chunks for each cell. Loading and storing data thus get the current index from the loop iteration rather than the GPU. The control code is shown in Figure 4.24b, along with the data index computations. As compression rarely occurs in lists, we omit its computation.

```
globalIndex = blockIndex * blockSize
                    + threadIndex

dataIndex = globalIndex   # Vector
dataIndex = 1             # Broadcast
dataIndex = <prefix sum> # Compressed
```

(a) Vector geometry.

```
cell  = globalIndex / listThreads
index = globalIndex % listThreads

while index < sizes[cell]:
    dataIndex = index # Vector
    dataIndex = 1     # Broadcast
    [...]
    index += listThreads
```

(b) List geometry.

Figure 4.24 – Thread-data assignment for each GPU kernel geometry.

Templates

Most array-based functions map easily to GPU code, as each thread in the computation is independent. We generate code statement-by-statement, translating each function in isolation using a templated approach, with type-correctness given by our PTX library. Named registers are allocated for each variable, and an unlimited number of temporaries for each template. We ignore register pressure, as the allocation

```
var t0:i32 = [...];
var t1:i32 = [...];
var t2:i32 = @plus(t0, t1);
```

(a) HorseIR input code.

```
.reg .s32 %t<3>;

add.s32 %t2, %t0, %t1;
```

(b) PTX output code.

Figure 4.25 – Example code generation template for adding two 32-bit integers.

scheme depends on the assembler. Shared memory is allocated as needed and capped at the device maximum. Common code generation templates include:

Element-wise: Unary and binary element-wise functions map to PTX instructions directly in nearly all cases. Shown in Figure 4.25 is an example translation for the sum of two 32-bit integers.

Date: Date and time functions are implemented according to the POSIX specification [102], and based on standard library implementations[2].

Reduction: Reductions aggregate vector data into a single value (e.g. `@min`). We follow the standard approach, implementing a warp-shuffle reduction to produce a single value per warp, followed by an atomic reduction operation (compare-and-swap for unsupported types [168]) to compute the overall value [138].

Compression: Compression follows a lazy approach, representing the result as a pair (variable, mask) and delaying the compaction. The mask is propagated to further computation, and only evaluated when required (e.g. storing or reducing data). Reduction considers only values with `true` in the mask, and store computes the write indexes using a prefix sum.

Member: Member function produces a boolean output, checking whether each value in one vector is present in another. Two implementations are provided: (1) an iterative approach that sequentially loops over the second vector using shared memory caching; and (2) an optimized hash table extended from the join implementation described in Section 4.3.4. Hash tables are used by default.

[2] https://www.sourceware.org/newlib/

Unique: Unique function produces a vector of distinct values. Two implementations are provided: (1) an iterative approach that loops over the vector; and (2) a library approach that sorts the input and removes duplicates using compression. We default to the iterative approach as it supports good performance for the entire benchmark. Additionally, our library strategy does not support finding unique elements in list cells due to incompatibility with `@each`.

4.3.4 Library

Library functions are implemented using one or more nested kernels, invoked by the runtime engine. Kernels are type-specific, and are therefore generated and compiled for each individual query according to a generic PTX pattern. While caching is possible for common types, compile-time generation is required for all other cases.

Sort: We implement bitonic mergesort, an in-place approach to sorting that iteratively sorts and merges power-of-2 sequences of increasing length [18]. Shared memory is used for sorting when the lengths are short enough as it greatly reduces the latency. Note that since HorseIR outputs the sorted data *by index*, we also maintain a list of indexes that are swapped along with the data as shown in Figure 4.26. For non-power of 2 data, we pad before sorting.

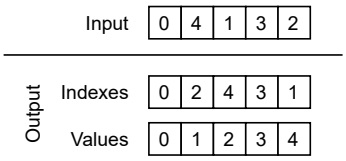


Figure 4.26 – Sort library function example.

Group: Grouping produces a dictionary, mapping each unique value (the "key") to its occurrences in the input data (the "values"), all of which are represented as indexes. An example group call is shown in Figure 4.27.

We follow a 3-step process building on the sort implementation:

1. Sort the data, returning sorted values and their indexes;

2. Find unique values, returning their indexes;
3. Build the dictionary (see Section 6.3).

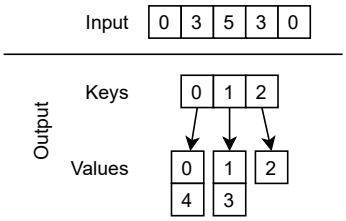


Figure 4.27 – Group library function example.

Unique values are found in parallel, checking the sorted data for changes in value, and applying the compression algorithm. Dictionary generation is described in a later section as it depends on the buffer allocation strategy.

Join: Joining two tables abstractly computes the cross product, only producing rows that conform to a predicate. We implement two alternative algorithms, a general purpose loop join, and an optimized hash join for equality joins. Both approaches follow the usual 3-stage process, as the join result is dynamic [92]:

1. Evaluate the cross product, computing the result size;
2. Allocate result buffers;
3. Re-evaluate the cross product, outputting the result.

Loop: The loop join algorithm computes the entire cross product `M x N`, with each thread producing the join result for one element in `M`. Optimization is possible using shared memory to coalesce reads and maximizing parallelism by selecting `M` as the larger table, though it remains an expensive algorithm. It remains useful, however, for general joins with non-hashable predicates.

Hash: We also provide an optimized hash join algorithm, based on a public domain GPU hash table [62] that uses the murmur3 hash function [12]. Each thread computes the result for a single element, using the hash table

to reduce the search space. While hashing is designed for equality, we can still use this approach for predicates which contain *at least* one equality comparison by only hashing on equality keys. Other keys are evaluated when probing the table. A similar approach is used in OmniSci [150].

Like: Strings are typically CPU resident and only transferred to the GPU as a hash value. String comparison is thus efficient and data transfers reduced. This is sufficient for most queries, but falls short for those using SQL `LIKE` as they require the string contents. We therefore employ a *string pad*, an array representation of all string data stored on the CPU with null termination. The hash values are therefore indexes into the pad and allow complex string operations on the GPU; collisions are thus impossible. Null termination is used in place of a length property. An example string pad and associated vector data is shown in Figure 4.28.

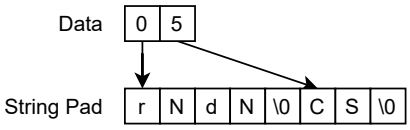


Figure 4.28 – `LIKE` string pad representation for two strings ("rNdN" and "CS").

Transferring the entire string pad to the GPU is costly, as it contains data for the entire database. We thus use an optimized *caching kernel* that transfers and caches strings required for the operation using *Unified Virtual Addressing* (UVA) – a seamless method for accessing CPU data allocated using CUDA [89]. Additionally, we can transfer multiple bytes (characters) in a single access using vector loads. We implement a simplified and optimized version of `LIKE` operation inspired by tutorials [57], adapted for the GPU and our representation.

4.3.5 Optimization

Code generation templates are efficient in isolation, with further optimization left to the assembler. We do however employ 3 basic optimizations important for queries:

1. Computing the GPU thread index is frequently used to access data. We therefore cache the computed index for all further computation;

2. Structured control-flow is used in all cases, greatly simplifying the assembler;

3. Power-of-2 computations (multiplication, division, remainder) used for sorting (and grouping) are optimized using bit operations.

4.4 Summary

Our frontend and compiler adapt query execution to the GPU environment, translating from high-level HorseIR programs to lower-level PTX intermediate code. We first parse the input query program and enforce semantic requirements, before automatically outlining parallel regions (i.e. kernels) using abstract shape and geometry information. Kernels are then compiled to efficient PTX code by composing intuitive algorithms for each operation, bypassing the first stage in the traditional compilation pipeline to reduce overhead. The output is further processed by the assembler described in Chapter 5 and executed in the runtime system presented in Chapter 6.

Chapter 5
Assembler

PTX intermediate code is a stepping stone that separates generation of GPU algorithms from low-level machine details. In this chapter we therefore present the backend stage 2 compiler, also referred to as the "assembler", that translates from PTX to a target specific binary. Designed for runtime systems, each step trades minor slowdowns in execution for major improvements in compilation, outperforming the proprietary NVIDIA pipeline on end-to-end workloads. Our balanced approach thus targets only necessary optimizations, and is significantly more lightweight than traditional ahead-of-time compilers like LLVM [127]. We begin by describing the compilation and analysis framework in Section 5.1, followed by register allocation in Section 5.2. The machine code specification is discussed in Section 5.3, along with code generation, control-flow structuring and peephole optimization. Lastly, we cover instruction scheduling in Section 5.4 and binary generation in Section 5.5 before summarizing our approach in Section 5.6. The resulting binary is loaded and executed by the runtime, presented in Chapter 6.

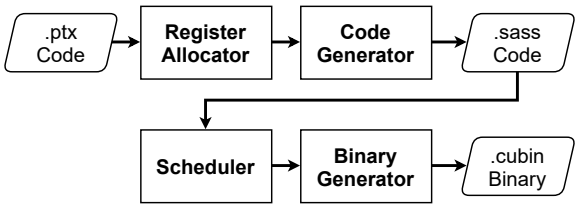


Figure 5.1 – rNdN backend compiler (assembler) architecture.

5.1 Framework

Similar to HorseIR, we implement a complete PTX analysis framework, supporting code generation and optimization; though as low-level language with unstructured control-flow, it requires additional steps. We therefore begin by constructing a control-flow graph (CFG), taking straight-line PTX code with labels and building a traversable graph of *basic blocks*, single-entry, single-exit sections of code. Following the standard approach, we traverse the program and collect *leading statements* [151]:

1. First statement;
2. Branch targets; and
3. Statements after branches.

Basic blocks are formed by collecting statements from one leader (inclusive) to another (exclusive). We standardize control-flow during this process by transforming predicated statements into traditional if (and if-else) branches using labels as shown in Figure 5.2. Predication is thus only used in conditional branching and simplifies the analysis framework. For efficiency reasons, predication is later restored in the machine code as described in Section 5.3.4.

```
@$p0 add.u32 $t2, $t0, $t1;
```

(a) Predicated statement.

```
@!$p0 bra End;
      add.u32 $t2, $t0, $t1;
End:
```

(b) Traditional if branch.

Figure 5.2 – Transforming predicated statements to a traditional if branches.

Building on the CFG, we implement a fixed-point analysis framework traversing basic blocks and collecting program properties. Basic blocks are traversed using a worklist algorithm, initialized with entry blocks and adding successors only if the analysis result changes [151]. The worklist is sorted using a postorder traversal for efficiency [180]. Similar to the HorseIR framework, we optimize the properties for each instruction using pointers and store only necessary data.

5.2 Register Allocation

PTX targets a virtual machine with unlimited registers, requiring a register allocation scheme when targeting hardware with finite resources. We describe GPU properties in Section 5.2.1, and the register allocation algorithm in Section 5.2.2.

5.2.1 GPUs

A modern GPU contains a large number of registers in each multiprocessor, partitioned between threads at runtime. As resources are finite, the number of registers per thread can therefore limit the *occupancy*, that is, "the ratio of active warps to the maximum number of warps supported on a multiprocessor" [164]. An efficient allocation must therefore balance spilling against occupancy, maximizing parallelism while not degrading performance of individual threads. Regardless of the allocation, each thread is limited to at most 255 general-purpose vector registers (`RX`) and 7 predicate registers (`PX`), in addition to the zero register (`RZ`) and true predicate (`PT`). Later architectures also support uniform registers for values shared across a warp computed using a special unit and instructions [30, 29, 106, 162].

General-purpose registers are all 32-bit in size, organized in banks as shown in Figure 5.3 [83]. Each instruction can read one value per bank in each clock cycle, with concurrent accesses to a single bank causing delays. We should therefore distribute source operands between banks where possible, or use scheduling directives discussed in Section 5.4 to cache reused registers. In our approach we choose to ignore bank conflicts, preferring instead to minimize compile time. As discussed with scheduling, queries are load/store-intensive and therefore less likely to benefit from small decreases in arithmetic latency than with more computation heavy applications.

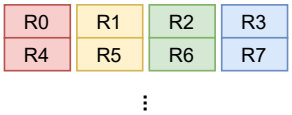


Figure 5.3 – Register organization with 4 banks, each shown in a different colour. The number of banks is architecture dependent [239].

For smaller data sizes, a single 32-bit register is allocated, or in the case of boolean data, a single predicate register. For larger data, consecutive registers are allocated in groups, aligned to the data size. An example 64-bit register pairing for long integer or double values is pictured in Figure 5.4 and can be extended to other types. Instructions access register pairings by referring to the lowest element.

(a) 32-bit register.

(b) 64-bit register pair (aligned).

Figure 5.4 – Register allocation for varying data sizes.

5.2.2 Linear Scan

Register allocation has been treated as a graph colouring problem, capturing variables that must be allocated to separate registers in an *interference graph* [151]. Each node represents a variable and each edge the existence of a program point where both variables are simultaneously live. Edges thus preclude register sharing and the best allocation corresponds to the lowest number of colours. Although minimizing registers appears optimal, NVIDIA forum posts describe the proprietary allocation scheme as using all "necessary [registers] to achieve the best performance"[1]. Preliminary experiments also show that small changes in register usage is mostly inconsequential.

As graph colouring is NP-hard, heuristics are often used to approximate the solution. Chaitin's algorithm is one such approach, iteratively pruning the interference graph and colouring nodes in the reverse order [34, 26]. This is suitable for offline compilation, but too costly for runtime systems. Instead, we use a *linear scan* allocator adapted for GPU properties [185, 219]. Like graph colouring it operates on liveness, but on a more conservative representation, *live intervals*, that assumes variables are continuously live from first to last use. Registers are then allocated on a first-come first-serve basis according to the following greedy algorithm. Note that we do not

[1] https://forums.developer.nvidia.com/t/on-the-register-allocation-optimization-of-cuda-compiler/69309/6

consider spilling heuristics and instead allocate as many registers as required as the current benchmark suite remains well below the limit (never exceeds 100 registers).

1. Compute live variables using a fixed-point data-flow analysis;
2. Transform live sets to live intervals `(start, end)`;
3. Sort live intervals by `start` position;
4. For each live interval `(start, end)`:
   (a) Free dead allocations (those with `end` position before the current `start`);
   (b) Allocate aligned register group `RX..RY`;
   (c) Add interval `(start, end, RX..RY)` to active allocations.

The resulting register allocation can be used to generate machine code. An example PTX program fragment and its associated register allocation steps are shown in Figure 5.5. We present the *outgoing* liveness set for each program point, capturing variables that are used in subsequent instructions. For linear scan, we picture the state at each iteration, with bold elements newly added and red indicating a discarded dead allocation. Note that PTX instructions use the following operand order:

```
destination, sourceA, sourceB, ...
```

5.3 Code Generation

We translate PTX code to SASS, a machine code representation that can be assembled to binary form. As with our first stage compiler, each pattern is designed and optimized in isolation. The SASS language is described in Section 5.3.1 and its associated memory layouts for each level in Section 5.3.2. Control-flow generation and branch inlining are discussed in Sections 5.3.3 and 5.3.4 respectively. We then cover code generation templates in Section 5.3.5 and peephole optimizations in Section 5.3.6.

```
1  ld.global.s32 %t0, [...];      {%t0}
2  ld.global.s32 %t1, [...];      {%t0, %t1}
3  add.s32 %t2, %t0, %t1;         {%t1, %t2}
4  add.s32 %t3, %t2, %t2;         {%t1, %t3}
5  add.s32 %t0, %t1, %t3;         {%t0}
6  st.global.s32 [...], %t0;      {}
```

(a) Live variables analysis (out sets).

(t0, 1, 5)
(t1, 2, 4)
(t2, 3, 3)
(t3, 4, 4)

(b) Sorted live intervals.

(t0, 1, 5)

Allocation	Active
t0 = R0	**(R0, 5)**

(t1, 2, 4)

Allocation	Active
t0 = R0	(R0, 5)
t1 = R1	**(R1, 4)**

(t2, 3, 3)

Allocation	Active
t0 = R0	(R0, 5)
t1 = R1	(R1, 4)
t2 = R2	**(R2, 3)**

(t3, 4, 4)

Allocation	Active
t0 = R0	(R0, 5)
t1 = R1	(R1, 4)
t2 = R2	(R2, 3)
t3 = R2	**(R2, 4)**

(c) Register allocation for each iteration, showing the newly allocated register and its associated interval `end` in bold. Intervals discarded as dead are shown in red.

Figure 5.5 – Linear scan register allocation example for PTX.

5.3.1 Target Language: SASS

SASS is a human readable representation of the machine code for NVIDIA GPUs. Varying between families, we support the Pascal and Ampere architectures directly, while other iterations may require additional tuning. Unlike PTX, which follows an open-source specification, SASS is officially closed-source with the exception of instruction mnemonics and disassembly tools for extracting code from assembled binaries [162]. Fortunately, several open-source projects have reverse engineered the instruction format, including asfermi for Fermi [246], MaxAs for Maxwell and Pascal [83], and TuringAs for Turing through Ampere [239]. A more systematic approach explored by Hayes et al. can also decode instructions for any architecture [90], and CuAssembler has also been proposed for multiple architectures [46]. Other low-level details have been discovered through microbenchmarks by Jia et al. and summarized in technical reports [107, 106]. Note that these projects are limited to manipulation of SASS code and are therefore not complete replacements of the stage 2 compiler.

While sufficient for Pascal, the decomposition of the Ampere instruction set has so far been limited and is insufficient to support SQL queries. We therefore analyze a large dump of instructions and their binary representations from CuAssembler, a proposed open-source assembler, reverse engineering the binary layout for each supported instruction [46]. This work, along with the open-source projects listed above and NVIDIA disassemblers form our understanding of SASS and low-level details. We show an example Pascal SASS program that increments an array in Figure 5.6a, the equivalent program for Ampere in Figure 5.6b and highlight differences below.

```
// Load block and local thread indexes
S2R R4, SR_CTAID.X ;
S2R R2, SR_TID.X ;

// Compute global thread index
//   c[0x0][0x8] - block size constant
XMAD.MRG R3, R4, c[0x0][0x8].H1, RZ ;
XMAD R2, R4, c[0x0][0x8], R2 ;
XMAD.PSL.CBCC R4, R4.H1, R3.H1, R2 ;

// Compute address using the thread
// offset and parameter constant
//   c[0x0][0x140] - low bits
//   c[0x0][0x144] - high bits
IADD R2.CC, R4, c[0x0][0x140] ;
IADD.X R3, RZ, c[0x0][0x144] ;


// Load, increment, store
LDG.E.U8 R4, [R2] ;
IADD32I R6, R4, 0x1 ;
STG.E.U8 [R2], R6 ;

EXIT ;
```

(a) Pascal code (sm61).

```
// Load block and local thread indexes
S2R R4, SR_CTAID.X ;
S2R R2, SR_TID.X ;

// Compute global thread index
//   c[0x0][0x0] - block size constant
IMAD R4, R4, c[0x0][0x0], R2 ;



// Compute address using the thread
// offset and parameter constant
//   c[0x0][0x160] - low bits
//   c[0x0][0x164] - high bits
IADD3 R2, P0, R4, c[0x0][0x160], RZ ;
IADD3.X R3, RZ, c[0x0][0x164], RZ,
    P0, !PT ;

// Load, increment, store
LDG.E.U8 R4, [R2.64] ;
IADD3 R6, R4, 0x1, RZ ;
STG.E.U8 [R2.64], R6 ;

EXIT ;
```

(b) Ampere code (sm86).

Figure 5.6 – SASS implementations to increment each element of an array. Each thread computes its position in the grid (i.e. the layout of all threads in the kernel), computes the offset variable address, and performs the increment.

Both instruction sets are expressed in 3-address notation, with instructions for arithmetic (integer and float), load/store, control-flow and synchronization. Differences primarily arise in binary representation and code generation templates. We note the following important differences between architectures:

Carry codes: Pascal uses flags (`.CC`/`.X`) to implement carrying, while Ampere uses a mix of explicit predicates (`P0`) and flags (`.X` only);

Arithmetic instructions: Code generation patterns for multiplication, division and remainder use instructions optimized for their architecture. Notably, Pascal uses a sequence of `XMAD` instructions (16-bit multiplier, 32-bit addition [59]) to implement merged 32-bit multiplication and addition, while Ampere supports efficient execution directly using `IMAD`.

Preferred instructions: For instructions which have multiple variants (e.g. `IADD`, `IADD32I`, and `IADD3`), we note a shift towards the more general version in newer architectures despite maintaining support for the entire set. We speculate this is due to an increased instruction size that allows merging previously distinct instructions and ease in code generation. In particular:

- Preference for 3-source instructions (e.g. `IADD3`) in Ampere[2];
- Preference for generalized instructions (e.g. `SHF` for shifting instead of `SHR` and `SHL`) in Ampere;

Memory layouts: Discussed in Section 5.3.2, the layout of constant memory spaces (i.e. `c[0x0][...]`) uses architecture-specific offsets for parameters.

5.3.2 Memory Hierarchy

Memory allocation and layout occurs both at compile time and during the linking phase. We discuss the organization of each level in the hierarchy in the following sections, with further details on the binary generation in Section 5.5.

[2] https://forums.developer.nvidia.com/t/ampere-sass-annotation/176758/6

Registers

General-purpose registers (`RX`) and predicates (`PX`) are the basic storage unit for computation, allocated using a GPU-adapted linear scan. Temporary registers used for computation are allocated as needed, subject to hardware thread limits. We omit uniform registers as they are not portable across all supported targets.

Special Registers

Special registers carry system parameters that vary between threads (e.g. indexes) or over time (e.g. clock). Immutable by the user-program, they are moved to the regular registers space using a special `S2R` instruction (**S**pecial **2** **R**egister). A subset of special registers is shown in Table 5.1 for both PTX and SASS. Note that GPU thread indexes are 3-dimensional, but we only use the x-dimension in our approach.

Table 5.1 – Special registers in PTX and SASS.

PTX Variable	Special Register
`%tid.x`	`SR_TID_X`
`%tid.y`	`SR_TID_Y`
`%tid.z`	`SR_TID_Z`
⋮	
`%globaltimer32_lo`	`SR_GLOBALTIMERLO`

Parameters

Kernel parameters are stored in a *constant space* accessible during execution. This includes explicit parameters in the function signature, as well as system parameters that are constant across threads and time. Parameters are stored at pre-defined offsets in the `0x0` constant space and represented in SASS as:

```
c[0x0][offset]
```

Dynamic system parameters are stored in the special registers space described previously. We show a subset of the parameters space in Table 5.2 along with the offsets

for both supported architectures. Explicit parameters are positioned after system parameters as they vary in quantity and size.

Table 5.2 – Constant parameter space layout for each architecture. `%ntid` variables indicate the size of each thread block dimension.

Variable	Pascal	Ampere
`%ntid.x`	`0x8`	`0x0`
`%ntid.y`	`0xc`	`0x4`
`%ntid.z`	`0x10`	`0x8`
⋮		
Explicit Parameters Offset	`0x140`	`0x160`

Constants

SASS instructions are constant size, either 64-bit or 128-bit depending on the architecture, with limited space for immediate values. A second constant space similar to parameters may thus store kernel-specific values, accessed in a similar manner:

```
c[0x2][offset]
```

Constant data is embedded directly into the binary by the assembler, and loaded by the CUDA runtime, requiring it be constant across all invocations. When generating operands, immediate values are used wherever possible, with constants only used when exceeding available bits. The *constant pad* is a concatenation of aligned values.

Shared/Global

Shared and global variables are relocatable objects allocated by the CUDA runtime, and are thus not part of the SASS definition. We discuss their address generation in Section 5.3.5 and linking directives in Section 5.5.

5.3.3 Structured Control-Flow

Each function body is represented as a graph of basic blocks (CFG). We therefore begin by describing the control-flow generation before covering templates for each instruction in Section 5.3.5. As this functionality is architecture-dependent, we describe each target separately after general control-flow structuring.

Recovering Structure

Supporting general unstructured control-flow proves challenging, particularly for the Pascal architecture. We simplify the problem by targeting only structured control-flow, enabling template-based code generation for structures equivalent to if-else branches, loops and breaks. This necessitates a well-structured CFG from the frontend compiler, limiting the kinds of control-flow used in code generation templates. Additionally, as a CFG has no inherent structure, we must recover each kind of supported control-flow. While these structures could have been directly generated in the earlier phase, we instead chose to recover structure and support more general input. We base our structuring algorithm on the following observations:

- Loops must have exactly one back edge, jumping from the *latch* to the *header*;
- Loop exit branches come in 2 varieties:
  - Latch blocks which conditionally break;
  - Intermediate blocks which conditionally break;
- If-else structures have two branches, each of which is independent;
- Structured control-flow reconverges at the *immediate post-dominator*, the first common block on all paths from the divergence point to the program exit; and
- **Blocks belong to a single structure in well structured control-flow.**

Each kind of control-flow structure is represented by a recursive data structure, storing its branches and the continuation point. The algorithms for recovering each kind of control-flow are based on control-flow analysis and described below [98].

We structure control-flow graphs recursively, greedily classifying basic blocks starting from the function entry point. Unstructured control-flow may thus be detected when a block belongs to more than one structure. For each block, we determine if it begins a loop, a well-structured break or if-else statement, or if it ends the current structure (e.g. branch of if-else statement, loop body). The main algorithm is shown in Figure 5.7, with each component described below.

```
def Structure(context, node):
    if node == None: // Base case
        return None

    // Recover loop structure with node as the header
    loopStructure = StructureLoop(context, node)
    if loopStructure != None:
        return loopStructure

    // Nodes reached from multiple unstructured paths not supported
    if processedNodes.contains(node):
        error("Unstructured control-flow")
    processedNodes.insert(node)

    // Get the reconvergence/continuation point of the current structure
    postDominator = ImmediatePDOM(node)

    // Divergent control-flow
    if GetOutDegree(node) == 2:
        trueBranch, falseBranch = GetBranches(node)
        return StructureBranch(context, trueBranch, falseBranch, postDominator)

    // Loop latch without condition ends body
    if context.kind == Loop and context.latch == block:
        return SequenceStructure(node, None)

    // Branch end point, continuation already handled by if-else structure
    if context.kind == Branch and context.exit == postDominator:
        return SequenceStructure(node, None)

    // Continuation point without branching
    nextStructure = Structure(context, postDominator)
    return SequenceStructure(node, nextStructure)
```

Figure 5.7 – Main algorithm for recovering control-flow structures.

Loops: Loops are structured recursively, first determining its properties (header, latch, body, exit) and then traversing the function body under a new loop context. We require that loops are well-nested with a single back edge, unique header, and well-structured breaks. If no back edges are detected, then the block does not begin a loop and processing continues in the main algorithm (`StructureLoop` returns `None`). Loop body blocks are identified by traversing the CFG in reverse, starting from the back edge and ending at the header, collecting all intermediate blocks. The immediate post-dominator of all body blocks serves as the loop exit, and is unique for each loop. Lastly, we recursively structure the CFG from the exit point to process the remaining program. The complete loop detection and structuring algorithm is shown in Figure 5.8.

```
def StructureLoop(context, node):
    // Basic block already detected as a loop header, continue in the main function
    if context.kind == Loop and context.header == node:
        return None

    // Consider all edges incoming to the node
    for predecessor in GetPredecessors(node):
        // Back edges are those who are dominated by the header
        if IsDominated(predecessor, node):
            // Decompose loop components
            header = node
            latch = predecessor
            body = GetLoopBody(header, latch)
            exit = GetLoopExit(header, body)

            // Recursively structure the loop body
            l_context = LoopContext(header, latch, exit, body)
            bodyStructure = Structure(l_context, header)

            // Structure exit point
            nextStructure = Structure(context, exit)
            return LoopStructure(bodyStructure, nextStructure)

    // No loop found
    return None
```

Figure 5.8 – Loop detection and structuring algorithm.

Loop exit: For loop structures we detect two kinds of exit structures. The first case corresponds to a tail-controlled loop, where we conditionally branch to the header or the exit block. The second a conditional break within the loop body, which incidentally covers head-controlled loops. Using the loop context, we can easily recover both kinds as they follow a standard structure. Unsupported cases are likewise easily detected, such as unstructured control-flow or continue. If no exit block is found, then the branch may correspond to a standard if-else statement. The complete loop exit algorithm is shown in Figure 5.9.

```
def StructureExit(context, trueBranch, falseBranch, postDominator):
    // Node (latch) branches to header and exit
    if node == context.latch:
        if trueBranch == context.exit and falseBranch = context.header:
            return ExitStructure(node, None)
        else if trueBranch == context.header and falseBranch == context.exit:
            return ExitStructure(node, None)
        error("Unstructured loop control-flow")

    // Break within loop body
    if trueBranch == context.exit and body.contains(falseBranch):
        falseStructure = Structurize(context, falseBranch)
        return ExitStructure(node, falseStructure)
    else falseBranch == context.exit and body.contains(trueBranch):
        trueStructure = Structurize(context, trueBranch)
        return ExitStructure(node, trueStructure)
    else if postDominator == context.header or postDominator == context.exit:
        error("Unstructured loop control-flow")

    // Standard if-else statement
    return None
```

Figure 5.9 – Loop exit detection and structuring algorithm.

If-else: If-else branch are constructed greedily, recursively structuring each branch under the assumption that paths are independent. Note that for branches without an else component, only one path is structured as the missing path points to the post-dominator. The structure is complete by processing the continuation point. We show the if-else structuring algorithm in Figure 5.10.

```
def StructureIfElse(context, trueBranch, falseBranch, postDominator):
    // Only keep branches within the structure
    if trueBranch == postDominator:
        trueBranch = None
    if falseBranch == postDominator:
        falseBranch = None

    // Recursively structure standard if-else
    l_context = BranchContext(postDominator)
    trueStructure = Structure(l_context, trueBranch)
    falseStructure = Structure(l_context, falseBranch)

    // Structure continuation point
    nextStructure = Structure(context, postDominator)
    return BranchStructure(block, trueStructure, falseStructure, nextStructure)
```

Figure 5.10 – If-else branch detection and structuring algorithm.

Pascal

In the Pascal architecture, instructions are issued by the warp scheduler using a single program counter per warp. This is trivial in SIMD when all threads within a warp execute the same path, but challenging when they *diverge*. As a SIMD unit may only execute a single instruction at a time, divergent paths are in effect serialized, executing one after the other and incurring additional cost [19]. Note that divergence between warps does not require any special handling as they operate on separate program counters. Control-flow is implemented using two methods:

- Instruction predication (`@P ...`); and
- Hardware divergence stack.

Predication associates instructions with a boolean mask, selectively enabling/disabling lanes in the SIMD unit on a per-instruction basis. This is effective for simple if-else branching, but is insufficient for more complex control-flow like loops and nested structures. In these instances, a hardware managed *divergence stack* may be used, managing active threads and their respective paths by tracking divergence and reconvergence points [47, 48, 154, 143, 124].

```
1         SSY L1 ;
2     @P  BRA L2 ;
3         IADD R0, R1, R2 ;
4         SYNC ;
5   L2:
6         IADD R0, R1, R3 ;
7         SYNC ;
8   L1:
```

(a) If-else statement in SASS using the Pascal divergence stack.

Line	Active Mask	Stack (Top on the left)
1	1111	
2	1111	<SYNC, L1, 1111>
3	0101	<DIV, L2, 1010> <SYNC, L1, 1111>
4	0101	<DIV, L2, 1010> <SYNC, L1, 1111>
5	1010	<SYNC, L1, 1111>
6	1010	<SYNC, L1, 1111>
7	1010	<SYNC, L1, 1111>
8	1111	

(b) Progress of the divergence stack as the program executes, capturing the state *before* each instruction executes. We assume only 4 threads for simplicity.

Figure 5.11 – Example if-else program using the divergence stack to manage divergence on Pascal (inspired by [47, 48, 124])

To manage complex control-flow, each warp in-flight has an active mask for currently enabled threads, and an associated hardware divergence stack. Tokens on the divergence stack represent execution paths and their associated *continuation points*. A continuation point is an address-mask pair that indicates the PC address and execution predicate once the current path is complete. By manipulating the stack, we can thus serialize divergent execution by selectively enabling and disabling threads and jumping between addresses. An example if-else program and its associated stack and active mask are shown in Figure 5.11.

Divergent sections begin by pushing a `SYNC` token onto the stack using an `SSY` instruction and its *reconvergence point*. Note that since we use only structured control-flow, the immediate post-dominator identified during structuring is ideal for reconvergence [67]. For top-level divergence the associated predicate mask is typically all `true` (1), but varies for nested control-flow. Next, a conditional branch pushes a `DIV` (divergence) token, storing the continuation point for the *path not-taken* and updating the active mask for the *path taken*. Execution continues along the taken path until complete and the stack is popped using a `SYNC` instruction. We can then update the active mask and PC, before executing the other branch. Once both paths are complete, the program continues from the reconvergence point.

```
      SSY L1;
  @!P SYNC ;
      <Body>
      SYNC ;
L1:
```

(a) If statement.

```
      SSY L1 ;
   @P BRA L2 ;
      <False>
      SYNC ;
L2:
      <True>
      SYNC ;
L1:
```

(b) If-else statement.

```
      SSY L1 ;
L2:
   @P SYNC ;
      <Body>
      BRA L2 ;
L1:
```

(c) While loop.

```
      SSY L1 ;
L2:
  @P1 SYNC ;
      <Body1>
  @P2 SYNC ;
      <Body2>
      BRA L2 ;
L1:
```

(d) While loop/break.

Figure 5.12 – Structured control-flow SASS code generation patterns for Pascal.

Given the divergence stack, we thus define code generation patterns for each kind of structured control-flow (continue is unsupported). Each template begins by declaring a divergence point and its associated reconvergence address, and using the stack to manage paths. We use one additional feature not covered in the previous example – predicated path termination (`@P SYNC`). This allows us to remove threads from the current active mask until all are inactive, at which time the reconvergence point is followed. Shown in Figure 5.12 are the templates for if-else structures and loops.

Ampere

Previous hardware iterations had a single PC per warp, requiring hardware assistance for intra-warp divergence. Beginning with Volta (precursor to Ampere), each thread maintains its own PC under a model named *Independent Thread Scheduling* [139]. This removes prior guarantees that threads within a warp execute the same instruction at the same time [171], but enables intra-warp divergence without the need for a dedicated stack. An instruction to force reconvergence and reinstate warp-level convergent execution guarantees, `WARPSYNC`, has correspondingly been added [162, 171]. Note that despite each thread managing a distinct PC, SIMD units are still limited to a single instruction at a time. Divergence within a warp thus still results in serialized execution and should be avoided wherever possible. Inter-warp divergence is unchanged from previous architecture designs.

With this change, the hardware stack has morphed to a *convergence barrier* implementation where state is associated with a barrier resource (max 16) instead of a stack token [162]. Although no longer necessary for most standard control-flow, it provides a method of selectively reconverging threads similar to the hardware stack. For simplicity, we choose to maintain patterns equivalent to those for Pascal, mapping each barrier resource to a position in the divergence stack. An example pattern for if statements is shown in Figure 5.13.

```
    BSSY B, L2
@!P BRA L1
    <Body>
L1:
    BSYNC B
L2:
```

Figure 5.13 – SASS code generation patterns for if statement on Ampere, using barrier resource `B` and reconvergence point `L2`.

5.3.4 Branch Inlining

Using the hardware divergence stack to manage divergence is costly and complex. We therefore prefer to use predication where possible, undoing the prior CFG transformation in select cases. In particular, we target if and if-else structures which have:

- No nested structures, each branch is a single basic block;
- No predicated instructions in either branch;
- No predicated code generation templates required for SASS.

Note that this depends on the contents of the basic blocks, as the equivalent SASS code must not have any predicated instructions. We also require that basic blocks are small as predicated instructions are issued even if no threads are active. A block size of 6 instructions was experimentally determined as the threshold.

5.3.5 Templates

Code generation is architecture specific and depends on its underlying instruction set. Like in the frontend compiler, we do not aim for optimal code, but rather a complete set of templates that are high-performance on query programs. Operand and instruction generation are discussed below, highlighting important differences between architectures where they arise.

Operands

Each instruction has a set of operands, specifying the input and output values for the operation. We support 4 kinds of operands in our implementation:

Registers/Predicates: Registers and predicates are given by the allocation scheme and temporary storage allocated as needed. If an immediate value requires register storage, the value is moved into a new temporary local to the instruction.

Addresses: Addresses are represented as a base-offset pair, where the base is a register and the offset either embedded directly in the instruction as an immediate or added to the base and stored in a temporary. Global and shared variables are generated as relocatable addresses, initially zero, which are patched when allocated. An example global variable 64-bit address is shown in Figure 5.14.

```
MOV R0, 0x0 ; // Address lo bits
MOV R1, 0x0 ; // Address hi bits
```

Figure 5.14 – Global variable address generation (shared addresses are 32-bit). The hex immediate values are later patched to the actual address.

Immediates: Immediate values are embedded directly in the instruction where possible, depending on the available bits. Floating point values are truncated, keeping the most-significant-bits and discarding zeros. On Pascal, values are typically limited to 20 or 32-bits, while Ampere supports 32-bits in all relevant cases. For larger data sizes, the kernel constants space is used.

Instructions

We adopt a simplistic code generation strategy, focusing on each instruction in isolation and translating directly from PTX to SASS with limited optimization beyond strength reduction. For complex instructions like multiplication, division and remainder we adopt optimized patterns from NVIDIA discovered using disassembly and rely on their internal expertise for performance [59]. Shown in Figure 5.15 are the platform specific code generation patterns for 32-bit unsigned integer multiplication.

```
// Input: R1, R2
// Output: R0
// Temporaries: R3, R4

XMAD.MRG R3, R1, R2.H1, RZ ;
XMAD R4, R1, R2, RZ ;
XMAD.PSL.CBCC R0, R1.H1, R3.H1, R4 ;
```

(a) Pascal code (`XMAD`).

```
// Input: R1, R2
// Output: R0

IMAD R0, R1, R2, RZ ;
```

(b) Ampere code (`IMAD`).

Figure 5.15 – Code generation templates for 32-bit unsigned integer multiplication.

We adopt NVIDIA's strategy of using general instructions instead of more specific versions in Ampere (e.g. `IADD3` instead of `IADD`), and consider platform-specific instruction specifications. As shown in Figure 5.16, Ampere code requires explicit predicates for the carry bit instead of flags used in Pascal (underlined for emphasis).

```
// Input: [R0, R1], [R2, R3]
// Output: [R4, R5]

IADD R4.CC, R0, R2 ;
IADD.X R5, R1, R3 ;
```

(a) Pascal code (carry flags).

```
// Input: [R0, R1], [R2, R3]
// Output: [R4, R5]
// Temporaries: P0

IADD3 R4, P0, R0, R2, RZ ;
IADD3.X R5, R1, R3, RZ, P0, !PT ;
```

(b) Ampere code (predicate carry).

Figure 5.16 – Code generation templates for 64-bit integer addition using carry codes.

5.3.6 Optimization

A full optimization framework is too costly for runtime compilation, owing to the fixed-point analyses required. We therefore focus on peephole optimizations that fix minor inefficiencies introduced during code generation – a more effective approach than implementing special case handling in each template. Shown in Figure 5.17 are the only two peephole patterns implemented in our system, both of which remove impactless instructions. We remove redundant moves when the source and destination are the same, and dead loads whose value is subsequently ignored. The former frequently occurs due to register allocation, and the latter from a lack of dead code elimination. More general approaches may be more effective, however, experiments with limited constant propagation and the associated register reduction did not significantly decrease execution. Additionally, the required fixed-point analyses notably increased compile-time and hurt overall performance.

```
MOV R2, R2 ;
```

(a) Redundant move.

```
LDG.E RZ, [R2] ;
```

(b) Dead load (zero register is immutable).

Figure 5.17 – Peephole optimizations for code generation artefacts, both of which safely remove impactless instructions without fixed-point analyses.

5.4 Scheduler

GPUs use compile-time instruction scheduling to satisfy dependencies and maximize instruction-level parallelism (ILP), addressing both correctness and performance. As a result, the hardware is simplified and program-specific optimization is enabled [83]. Our scheduler is based on MaxAs, optimized for our context and extended across multiple platforms [83]. We describe the scheduling directives in Section 5.4.1 and the associated instruction classes for each hardware architecture in Section 5.4.2. The scheduling algorithm and key motivators are presented in Sections 5.4.3 and 5.4.4, and an optional optimization for variable-cycle instructions in Section 5.4.5.

5.4.1 SCHI Directives

Scheduling directives are embedded in the binary program, either as part of the instruction format or interwoven in the instruction stream [106, 90, 83]. Also called SCHI directives, recent architectures like Ampere use the former, while Pascal has one compound directive for every 3 instructions. We show the binary format in Figure 5.18 and describe each component below.

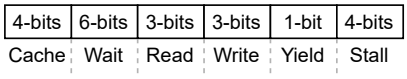


Figure 5.18 – SCHI directives binary layout.

Stall count: After each instruction is issued, a fixed-length stall delays further execution of the warp before it becomes re-eligible for dispatch. This allows fixed-cycle instructions (e.g. `IADD`, `MOV`) to satisfy dependencies.

Yield flag: Each cycle, the warp scheduler may either continue with the current warp or switch to another. The yield flag serves as a hint to the hardware [83, 239].

Read/write barriers: For variable-cycle instructions (e.g `LDG`, `STG`) barrier resources allow warps to wait an indeterminate length before continuing. A set of 6 barrier resources shared between reads and writes tracks the number of unsignaled dependencies using a *scoreboard register* [108]. Setting a barrier increases the scoreboard after dispatch, while signaling decreases.

Wait barriers: Instructions may wait on a dependency barrier before execution (previously set using read/write barriers). All specified scoreboard registers must reach zero before the instruction is issued.

Register cache flags: Register bank conflicts serialize accesses and slow execution. The hardware thus provides a small cache, one per source operand position, to store recurring registers. Bank conflicts can be avoided by accessing the cache.

An efficient schedule must therefore minimize barrier waits and stall counts to increase ILP and decrease cycles waiting. Register caching and yield hints are optional.

5.4.2 Instruction Classes

We associate each instruction with an *instruction class* that determines its execution properties. While NVIDIA provides some limited information, we recovered the bulk of the specification for Pascal from MaxAs classes [83] and updated this information with our own microbenchmarks for Ampere. Instruction classes (collectively a *profile*) allow our scheduler to make correct and efficient instruction orderings. We summarize profiles for Pascal and Ampere in Table 5.3 and describe each property below.

Functional unit: Each instruction is executed on a functional unit, corresponding to the hardware defined in Chapter 2. The functional unit depends on the operation, as well as the data type. For Pascal, we use the generic "Core" notation for the main functional units, whereas Ampere is type-specific.

Depth: Fixed latency between dispatch and result availability for fixed-cycle instructions, or until the barrier resource becomes active for variable-cycle instructions.

Write: Variable latency between dispatch and destination register write for variable-cycle instructions, taken as an approximation for all instructions in the class.

Read delay: Fixed latency between dispatch and time at which the source register is read, allowing ILP with the source computation.

Read hold: Variable latency between dispatch and time at which the source register is available for reuse for variable-cycle instructions.

Throughput: Determined by the number functional units, throughput latency is given by the number of cycles required to dispatch all threads in a warp [168].

Dual issue: If dual issue is supported (Pascal only), two instructions taking different data paths may be issued in a single cycle. We follow the MaxAs convention and specify if dual issue is supported for the second instruction.

Reuse cache: Only certain instructions support the register operand cache.

Note that these properties are approximations used to guide a correct schedule, but may differ from the actual hardware implementation.

Table 5.3 – Scheduling properties for each class. Variable latencies (indicated by *) are approximate and used only as hints, whereas fixed latencies are exact. Throughputs are computed from the number of functional units and the CUDA guide [168].

Class	Unit	Latencies					Dual	Reuse
		Depth	Write*	Read Delay[a]	Read Hold*	Throughput		
S2R	S2R	2	~25	-	-	4	✗	✗
Control	Core	5	-	-	-	N/A[b]	✗	✗
Integer	Core	6	-	-	-	1	✗	✓
Single	Core	6	-	-	-	1	✗	✓
Double	DP	2	~45	-	~10	32	✗	✓
Special	SFU	2	~13	-	~10	4	✓	✗
Comparison	HCore[c]	13	-	-	-	2	✗	✓
Shift	HCore	6	-	-	-	2	✗	✓
Shared Load	LDST	2	~25	2	~8	4	✓	✗
Shared Store	LDST	2	-	2	~8	4	✓	✗
Global Load	LDST	2	~165	4	~13	4	✓	✗
Global Store	LDST	2	-	4	~15	4	✓	✗

(a) Pascal instruction properties. Adapted from MaxAs [83].

[a]Read delay for predicates is always zero.
[b]Control instructions must complete.
[c]HalfCore, executes at half throughput without impacting latency of other instructions.

Class	Unit	Latencies					Reuse
		Depth	Write*	Read Delay	Read Hold*	Throughput	
S2R	S2R	2	~25	-	-	8	✗
Control	F32	4	-	-	-	N/A[a]	✗
Integer	I32	5[b]	-	-	-	2	✓
Single	F32	5	-	-	-	1	✓
Double	DP	2	~45	-	~12	64	✓
Special	SFU	2	~18	-	~15	8	✗
Comparison	I32	13	-	-	-	2	✓
Shift	I32	5	-	-	-	2	✓
Shared Load	LDST	2	~25	-	~8	8	✗
Shared Store	LDST	2	-	-	~9	8	✗
Global Load	LDST	2	~220	-	~10	8	✗
Global Store	LDST	2	-	-	~14	8	✗

(b) Ampere instruction properties. Dual issue unsupported.

[a]Control instructions must complete.
[b]Integer instructions may have lower latency in some circumstances, omitted for our scheduler.

5.4.3 Scheduler Properties

Our scheduler must be correct and efficient, producing an instruction order that reduces stalls and maximizes ILP while satisfying dependencies. We therefore define 5 key properties of our approach, each of which is shown by example in Figure 5.19:

1. Fixed-latency instructions **must** complete before their result is accessed;
2. Variable-latency instructions **must** set the appropriate read/write barrier, and their children **must** wait until signalled;
3. Independent instructions **should** be executed concurrently, increasing ILP;
4. Long critical paths **should** be prioritized, hiding their execution with ILP;
5. Instruction throughput **should** be considered, delaying execution of further instructions on the same functional unit (not pictured in Figure 5.19).

Note that although throughput may impact instruction dispatch, the hardware will insert stalls internally [83], thus the scheduler only considers it a hint. We do not consider dual issue and register reuse in our scheduler as initial experiments showed that the compilation cost exceeds benefits. This is likely due to the intensive read/write nature of queries, resulting in a program dominated by high-latency operations.

```
IADD32I R0, R0, 0x1 ; (Stall=6)
IADD R0, R0, R1 ;     (Stall=...)
```

(a) Fixed-latency instructions.

```
LDG R0, [R1] ;         (Write=SB0)
IADD32I R0, R0, 0x1 ; (Wait=SB0)
```

(b) Variable-latency instructions.

```
IADD32I R0, R0, 0x1 ; (Stall=1)
IADD32I R2, R2, 0x1 ; (Stall=5)
IADD R0, R0, R0 ;     (Stall=...)
```

(c) Independent instructions.

```
LDG.E R0, [R2] ;
LDG R1, [R0] ;
IADD32I R1, R1, 0x1 ;
```

(d) Long critical path.

Figure 5.19 – Key instruction scheduling properties for correctness and efficiency.

5.4.4 Scheduler Algorithm

Instruction scheduling produces an efficient and correct instruction order, typically processing a dependency graph that captures relationships between instructions. We schedule each function at a basic block level, further divided into *schedulable sections*. Each section is free of control dependencies, either from control-flow or instructions that prevent reordering (e.g. clock reads). The associated dependency graphs consist only of data dependencies and are thus smaller and more efficient to construct and analyze. We implement a variant of list scheduling [75], traversing each schedulable section's dependency graph and selecting the next instruction based on a heuristic. The set of instructions eligible for scheduling are those whose dependencies may be satisfied, either fixed or variable latency. Our heuristic selects the next instruction based on 3 ordered properties, reducing stall counts and maximizing ILP:

1. Prefer **lower** expected latency before dependencies are satisfied;
2. Prefer **longer** critical path length to end of program; and
3. Prefer **lower** line number.

Expected latencies are given by the fixed and variable-cycle latencies of each parent instruction, specific to the hardware profile. We compute the critical path length in a similar manner, considering the expected latency of each instruction in the path. The line number is used to break ties and produce a deterministic schedule. List scheduling is also employed by MaxAs [83], from which we derive our approach, though we incorporate dependency barriers and employ an alternative heuristic.

Algorithm

List scheduling is an iterative algorithm, maintaining the *clock* of the last scheduled instruction, the *available time* of each instruction, and a list of currently *available instructions*. It begins by computing a dependency graph for the schedulable section, computing write-read (true/flow), read-write (anti) and write-write (write) dependencies, and initializing the list of available instructions as those with no predecessors.

Intuitively, available instructions are those whose dependencies are satisfied and have available time less than or equal to the current clock cycle. In each iteration, we then heuristically select the next instruction from the available set, schedule the current instruction, and compute the earliest available time of each successor. Scheduling an instruction adds it to the current schedule, sets the stall count, and manages dependency barriers. A high-level overview of the list scheduling algorithm is shown in Figure 5.20. We describe barrier selection for variable-cycle dependencies in Section 5.4.5 and stall count selection below.

```
def ListSchedule(section):
    clock = 0
    schedule = []
    activeBarriers = []
    availableTimes = []
    availableInstructions = []

    // Initialize dependency graph, and set of available instructions
    BuildDependencyGraph(section)
    InitializeAvailableInstructions(section)

    // Select the next instruction according to the heuristic
    while [previousInstruction, nextInstruction] = Next(availableInstructions):
        schedule.insert(nextInstruction)

        // Manage read/write barriers
        InsertWaitBarriers(nextInstruction)
        InsertReadWriteBarriers(nextInstruction)

        // Reduce the previous instruction stall while satisfying dependencies
        UpdateStall(previousInstruction)

        // Update earliest available time of all successors
        UpdateAvailableTimes(availableTimes)

        // Update the list of available instructions and heuristics
        UpdateAvailableInstructions(availableInstructions)

    // Ensure all instructions are complete
    WaitOnBarrier(activeBarriers)
```

Figure 5.20 – Instruction scheduling algorithm.

Stall Counts

Setting the stall count for an instruction is a two step process, first selecting an initial value which ensures completion of the current section and then optimizing the smallest gap which satisfies dependencies. At the end of the section we thus guarantee that all fixed-latency instructions are complete, while variable-latency instructions require waiting on all active barriers. The minimum stall count is the difference between the available time and the current clock cycle, with the available time for each dependency kind shown in Table 5.4, adapted from MaxAs for true and anti-dependencies [83] and augmented with write dependencies. For our heuristic we also maintain the *expected* time to account for variable-latency dependencies and (not pictured) throughput.

Table 5.4 – Available and expected stall counts for each dependency kind.

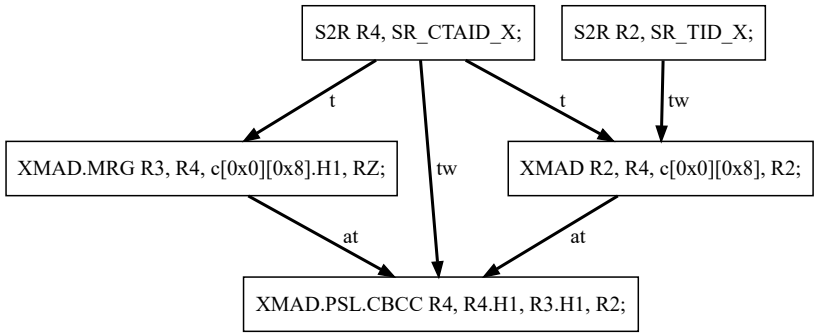
Kind	**Variable-Latency**		**Fixed-Latency**
	Available	**Expected**	
Write-Read (True)	Depth latency (2)	Write latency	Depth latency parent - read latency child
Write-Write (Write)	Depth latency (2)	Write latency	Depth latency parent - depth latency child
Read-Write (Anti)	Depth latency (2)	Read hold	1

In particular note that variable-latency instructions must stall 2 cycles for the barrier to become active [83]. For fixed-latency write dependencies, depth latencies of both the parent and child instructions are adjusted by their respective throughputs, ensuring that writes are in-order for all threads. For fixed-latency anti-dependencies, we assume the read latency of the parent is smaller than the write latency of the child.

Example

An example schedule using the Pascal profile is shown in Figure 5.21, continuing the running example for the chapter. Its associated dependency graph has been trimmed for simplicity, showing only the initial instructions and their dependencies. Dependencies include true dependencies (t), anti-dependencies (a), and write dependencies (w). Instruction schedules are formatted as a tuple:

[Cache, Wait barriers, Read barriers, Write barriers, Yield flag, Stall count]



(a) Dependency graph for a subset of the example. Edges are tagged with their dependencies: t = true/flow dependency, w = write dependency, a = anti-dependency.

```
S2R R4, SR_CTAID_X;                          [C-;B-;R-;W2;Y1;S1]
S2R R2, SR_TID_X;                            [C-;B-;R-;W2;Y1;S2]

XMAD.MRG R3, R4, c[0x0][0x8].H1, RZ;         [C-;B2;R-;W-;Y1;S1]
XMAD R2, R4, c[0x0][0x8], R2;                [C-;B-;R-;W-;Y1;S6]
XMAD.PSL.CBCC R4, R4.H1, R3.H1, R2;          [C-;B-;R-;W-;Y1;S6]

IADD R2.CC, R4, c[0x0][0x140];               [C-;B-;R-;W-;Y1;S6]
IADD.X R3, RZ, c[0x0][0x144];                [C-;B-;R-;W-;Y1;S2]

LDG.E.U8 R4, [R2];                           [C-;B-;R5;W0;Y1;S2]
IADD32I R6, R4, 0x1;                         [C-;B0;R-;W-;Y1;S2]
STG.E.U8 [R2], R6;                           [C-;B-;R5;W-;Y1;S4]

EXIT;                                        [C-;B5;R-;W-;Y1;S5]
```

(b) Scheduled SASS code, using notation from an offline blog[a], inspired by MaxAs [83]. For each instruction, we tag the: [Cache, Wait barriers, Read barriers, Write barriers, Yield flag, Stall count].

[a] https://newspanning.com/article/179/ (inaccessible, blog offline)

Figure 5.21 – Scheduled SASS function for Pascal.

5.4.5 Barriers

Dependency barriers govern variable-latency dependencies between instructions, with the parent setting the barrier and child waiting until signaled. Implemented using a scoreboard register, each barrier resource counts the number of in-progress instructions and signals dependencies only once all are complete. This is problematic as the number of barriers is finite, 6 on current hardware, and the number of independent variable-latency instructions executed concurrently may be higher. NVIDIA addresses this issue by providing a special instruction, `DEPBAR`, which waits only until the barrier scoreboard register decreases below a threshold. By counting the number of variable-cycle instructions sharing the barrier between the parent and child, we can determine the required value, shown by example in Figure 5.22. Note that instructions sharing the same scoreboard must therefore complete in-order.

```
LDG.E R0, [R4] ; (Write=SB0, Count=1)
LDG.E R1, [R6] ; (Write=SB0, Count=2)
LDG.E R2, [R8] ; (Write=SB0, Count=3)

DEPBAR.LE SB0, 0x2 ;   // Wait until SB0 barrier has at most 2 elements
IADD32I R0, R0, 0x1 ;
```

Figure 5.22 – Scoreboard registers allow a single barrier to be shared for concurrent variable-latency instructions without waiting for all to complete.

Our scheduler inserts partial barriers where possible, increasing ILP from variable-latency instructions. As signaling must occur in order, we use scoreboards only for the most expensive instructions, one for each of reads and writes to global and shared memory (4 total). If we must wait until the scoreboard reaches zero, the typical SCHI directives are used, otherwise we prefer partial barriers. All other instruction classes (e.g. double precision) share the remaining two barrier resources, one for reads and one for writes, with dependencies handled through SCHI directives. Note that NVIDIA has a strategy outlined in their patent for further sharing [108]. Despite their presence in Ampere, we have been unable to reliably use partial barriers for optimization – likely due to re-ordering changes that are not publicly available.

5.5 Binary Generation

Lastly, we translate the SASS program to an assembled binary that is loaded and executed by the CUDA runtime. The assembly stage building the binary program is described in Section 5.5.1 and ELF file generation in Section 5.5.2.

5.5.1 Assembly

Assembling a program translates from a human readable representation to binary form. We build on our SASS library, generating a binary instruction sequence and associated metadata for each function according to the following steps:

1. Sequence basic blocks, creating a linear sequence of instructions;
2. Add termination self-branch, signaling the end of function;
3. Pad instruction sequence to a multiple (Pascal: 6; Ampere: 16) using `NOP`s;
4. Insert SCHI directives (Pascal only, Ampere uses inline directives);
5. Resolve branch target addresses;
6. Construct function metadata:
   - Global/shared variables and relocations;
   - Constant memory pad;
   - Synchronization barrier count;
   - CRS stack size (divergence stack);
   - Register count;
   - (Optional) Required thread dimensions;
   - Indirect branches (`SSY`);
   - Addresses of some special instructions: `EXIT`, `SHFL` and certain `S2R`;
7. Translate instruction sequence to binary (Pascal: 8 byte; Ampere: 16 byte).

Metadata is used for NVIDIA-specific sections of the assembled ELF file, directing the runtime. Note that for Ampere, an additional two registers are required per thread for the program counter[3]. The assembled example program used throughout the chapter is shown in Figure 5.23, with each instruction associated with its address and binary format. Blank lines correspond to SCHI directives, while an additional padding multiple shows the format of `NOP` instructions.

```
// Parameters: 0x8 bytes
// Registers: 5
// Barriers: 0
// S2R CTAID Offsets: 0x0008
// EXIT Offsets: 0x0070

/* 0x0000 */                                                  /* 0x009fc400ea400751 */
/* 0x0008 */    S2R R4, SR_CTAID_X;                           /* 0xf0c8000002570004 */
/* 0x0010 */    S2R R2, SR_TID_X;                             /* 0xf0c8000002170002 */
/* 0x0018 */    XMAD.MRG R3, R4, c[0x0][0x8].H1, RZ;          /* 0x4f107f8000270403 */
/* 0x0020 */                                                  /* 0x001fd800fec007f6 */
/* 0x0028 */    XMAD R2, R4, c[0x0][0x8], R2;                 /* 0x4e00010000270402 */
/* 0x0030 */    XMAD.PSL.CBCC R4, R4.H1, R3.H1, R2;           /* 0x5b30011800370404 */
/* 0x0038 */    IADD R2.CC, R4, c[0x0][0x140];                /* 0x4c10800005070402 */
/* 0x0040 */                                                  /* 0x003fc800a24007f2 */
/* 0x0048 */    IADD.X R3, RZ, c[0x0][0x144];                 /* 0x4c1008000517ff03 */
/* 0x0050 */    LDG.E.U8 R4, [R2];                            /* 0xeed0200000070204 */
/* 0x0058 */    IADD32I R6, R4, 0x1;                          /* 0x1c00000000170406 */
/* 0x0060 */                                                  /* 0x001ffc20fea005f4 */
/* 0x0068 */    STG.E.U8 [R2], R6;                            /* 0xeed8200000070206 */
/* 0x0070 */    EXIT;                                         /* 0xe30000000007000f */
/* 0x0078 */    BRA `(_END) [0x0078];                         /* 0xe2400fffff87000f */
/* 0x0080 */                                                  /* 0x001f8000fc0007e0 */
/* 0x0088 */    NOP;                                          /* 0x50b0000000070f00 */
/* 0x0090 */    NOP;                                          /* 0x50b0000000070f00 */
/* 0x0098 */    NOP;                                          /* 0x50b0000000070f00 */
/* 0x00a0 */                                                  /* 0x001f8000fc0007e0 */
/* 0x00a8 */    NOP;                                          /* 0x50b0000000070f00 */
/* 0x00b0 */    NOP;                                          /* 0x50b0000000070f00 */
/* 0x00b8 */    NOP;                                          /* 0x50b0000000070f00 */
```

Figure 5.23 – Assembled SASS program for Pascal.

[3] https://stackoverflow.com/questions/47535903/register-consumption-of-per-thread-program-counters-in-volta

5.5.2 ELF Files

The CUDA runtime can load relocatable ELF files, allowing further linking against system libraries like `libdevice`. We construct a CUDA-compliant ELF file directly in memory using an open source library, ELFIO [126]. Each section is reverse-engineered from microbenchmarks using NVIDIA binary tools [162], varying program properties to determine the expected result. Existing open-source assemblers like MaxAs and TuringAs [83, 239] were also used as guides but proved too limited for a complete implementation (e.g. relocations, global variables). Sections include the header and global variables as well as function-specific sections for code, constants, metadata, shared variables and relocations. We provide a high-level description of each section below, translating from the assembled binary and metadata to executable form.

Header

A CUDA-compliant ELF file targets the NVIDIA architecture, specified in the header section. Shown in Table 5.5 are the required properties in addition to the standard ELF attributes not pictured. We include the target streaming multiprocessor version (SM) as the instruction format is architecture dependent and required for execution.

Table 5.5 – Header section properties.

Attribute	Value
OS ABI	`ELFOSABI_NVIDIA`
ABI Version	`ELFABI_NVIDIA_VERSION`
Version	`ELF_CUDA_VERSION`
Machine	`EM_CUDA`
Flags	`EF_CUDA_VIRTUAL_SM EF_CUDA_SM EF_CUDA_TEXMODE_UNIFIED EF_CUDA_64BIT_ADDRESS`

.nv.global

Global variables accessible by all functions are defined in a single section large enough for all allocations. Each variable is entered into the symbol table as a global CUDA object, specified with their size and offset in the section. For natural alignment of the offset, we sort variables by decreasing data size.

.nvinfo

Function metadata is split between two sections: a global `.nvinfo` section and a further function-specific section. The rational behind the division remains unclear. Each attribute specifies an execution property or memory allocation. Execution properties include thread block dimensions and addresses of `EXIT` instructions, while memory allocations specify the number of registers and the layout of the kernel parameter space. The complete list of supported properties is listed in Table 5.6, divided between the global and function-specific sections. We additionally support suspected compatibility/versioning attributes that vary between architectures: `EIATTR_SW2393858_WAR` and `EIATTR_SW1850030_WAR` for Pascal, `EIATTR_SW2861232_WAR` for Ampere.

Table 5.6 – Function metadata sections.

Attribute	Description
`EIATTR_REGCOUNT`	Number of registers allocated per thread
`EIATTR_MAX_STACK_SIZE`	*Unused* (`0x0`)
`EIATTR_MIN_STACK_SIZE`	*Unused* (`0x0`)
`EIATTR_FRAME_SIZE`	*Unused* (`0x0`)

(a) `.nvinfo` section properties.

Attribute	Description
`EIATTR_CUDA_API_VERSION`	`ELF_CUDA_VERSION`
`EIATTR_PARAM_CBANK`	Layout of kernel parameters space
`EIATTR_CBANK_PARAM_SIZE`	Size of user-defined kernel parameters
`EIATTR_KPARAM_INFO`	Layout of user-defined kernel parameters, one per parameter
`EIATTR_MAXREG_COUNT`	Maximum registers per thread (may be different than allocated registers)
`EIATTR_INDIRECT_BRANCH_TARGETS`	Addresses and targets of indirect branch `SSY` instructions (Pascal only)
`EIATTR_S2RCTAID_INSTR_OFFSETS`	Addresses of `S2R` instructions accessing `CTAID` thread block dimension
`EIATTR_EXIT_INSTR_OFFSETS`	Addresses of `EXIT` instructions
`EIATTR_COOP_GROUP_INSTR_OFFSETS`	Addresses of `SHFL` instructions
`EIATTR_REQNTID`	Required thread dimensions
`EIATTR_MAX_THREADS`	Maximum thread dimensions
`EIATTR_CTAIDZ_USED`	`CTAID` thread block `Z` dimension accessed
`EIATTR_CRS_STACK_SIZE`	Size of divergence stack (Pascal only)

(b) `.nvinfo.function` section properties.

.text.function

A typical `.text` section gives the assembled function body, each instruction represented in binary as either 8-byte for Pascal or 16-byte in Ampere. We include two section attributes for resource allocation:

- Flag `SHF_BARRIERS`: Number of synchronization barriers; and
- Info `SHI_REGISTERS`: Number of registers (duplicated in metadata).

A new record is added to the symbol table serving as a CUDA runtime entry point.

.nv.constant0.function

Kernel parameters are specified in the metadata and allocated as a zero-initialized section. Its size is given by architecture-specific system parameters and the signature.

.nv.constant2.function

Constant values too large for the instruction format are concatenated into a constants pad during code generation, one per function, included directly in the ELF file.

.nv.shared.function

Each function contains a shared section, large enough to contain all shared variables. Variables are entered into the symbol table as a shared CUDA object and the offset computed by the linker (unlike global variables which pre-compute their offset).

.rel.text.function

Global and shared variables are loaded from relocatable addresses using `MOV` instructions from code generation. Relocations indicate each such instruction to the linker along with the required patching once the variable is allocated. Note that relocations are architecture specific as they depend on the instruction format. We show each kind of relocation in Table 5.7. Note that global addresses are 64-bit and thus split into two components: low bits and high bits. The format of each relocation is as follows:

`R_CUDA_ABS{size}_{position}`

Size indicates the number of bits used for the address (possibly with `LO` or `HI` qualifiers), and position is the offset within the patched instruction.

Table 5.7 – `.rel.text.function` section address relocations.

Address	Pascal	Ampere
Global variable	`R_CUDA_ABS32_LO_20` / `R_CUDA_ABS32_HI_20`	`R_CUDA_ABS32_LO_32` / `R_CUDA_ABS32_HI_32`
Shared variable	`R_CUDA_ABS24_20`	`R_CUDA_ABS32_32`

5.6 Summary

The assembler, or backend compiler, translates intermediate PTX code to the target architecture instruction set. Presenting the first detailed look at an end-to-end compilation pipeline for NVIDIA GPUs, we define a balanced approach suitable for short-running queries by trading minor execution slowdowns for major compilation speedups. Our approach allocates registers, generates machine instructions, and performs limited optimizations including instruction scheduling and peephole optimization. The resulting binary is compatible with the CUDA runtime – the details of which are presented in Chapter 6.

Chapter 6
Runtime

The runtime system is responsible for query execution, taking the CPU control-code and assembled CUDA binary as input. It loads information from the database, performs the computation and returns the result to the user. We formalize the necessary components that support efficient execution of compiled queries on the GPU, including the interpreter, SQL library and GPU engine in Section 6.1 followed by the supporting CUDA runtime and associated libraries in Section 6.2. Lastly, we discuss efficient data management techniques in Section 6.3, covering representation, allocation, transfer and optimizations, before summarizing in Section 6.4.

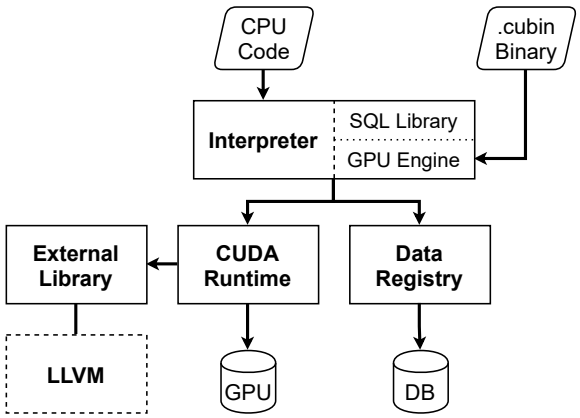


Figure 6.1 – rNdN runtime architecture.

6.1 Interpreter

Query programs are split between the CPU and GPU, the former reserved for functions like string manipulation and data layout, and the latter for outlined kernels. We implement a simple interpreter, executing HorseIR functions in a virtual environment and storing immutable data for each variable. CPU functions are evaluated directly, while GPU kernels and libraries are passed to their execution engines. We describe library functions in Section 6.1.1 and the GPU execution engine in Section 6.1.2.

6.1.1 SQL Library

Library function calls reference a set of nested functions, executed using custom control-flow and data allocations not present in HorseIR. We highlight the following examples, but also support `@unique` by sorting and `@member` by hash tables.

Sort: Sorting is implemented via bitonic mergesort described in Section 4.3.4, iteratively sorting the data in-place into bitonic sequences of increasing size. Each library call provides both global and shared memory variants, and the runtime selects the implementation. Shared memory is preferred where possible due to its efficiency. The complete algorithm is as follows:

1. Pad input data size to a power-of-2 (required by bitonic sort);
2. Iteratively sort in-place; and
3. Resize output data to remove padding.

Group: Grouping builds a key-value dictionary, collecting data with common attributes into bins. Extending the sort library, input data is first sorted before determining unique elements with a further nested kernel. The output dictionary is constructed using a pre-compiled system provided library kernel and optimized storage layout discussed in Section 6.3.1.

Join: Following the typical 3-step process described during code generation, each join library call references two nested kernels: one for computing the size and a

second for the actual join. Allocations occur automatically before invoking the second kernel using the GPU execution engine. The selection of hash vs. join is done at compile time as it depends on the join predicate - a static property.

Like: String data is represented as an index into a string pad, with each entry a unique value. We first update the GPU string cache using a special vector-loading kernel that pulls only required data, and then execute the like operation. Both kernels are pre-compiled by the runtime as they are type and data agnostic.

6.1.2 GPU Engine

The GPU execution engine initializes and invokes GPU kernels, setting up input and output data buffers along with thread layouts. We follow a 6-step process, used for both outlined kernels and previously described nested library functions:

1. Compute runtime geometry;
2. Transfer input buffers;
3. Initialize return buffers;
4. Execute kernel; and
5. Resize return buffers.

The runtime thread geometry is computed by shape and geometry analyses described in Chapter 4. However, we can now use exact values rather than symbolic representations for buffer sizes. Input buffers are transferred, while return buffers are allocated and initialized as required. Buffer initialization may either set an initial value (e.g. the null value for reductions), or copy existing data when only modifying a subset (e.g. scatter/indexed write). For compressed output buffers we conservatively allocate their maximum size and resize after execution. In most cases, resizing only updates the size property of the buffer as the cost of reallocating and copying data is prohibitively expensive. Note that this strategy is not possible for join outputs, as the required buffer size is the product of the input sizes and would be excessively large. Kernels may be launched with dynamic shared memory if needed.

Thread Layouts

Given the abstract kernel geometry, we compute the exact size of each thread block and the number of blocks required to compute the entire result. Note that since each thread block must have identical size (required by CUDA), unused threads are disabled by bounds checking in the generated code. Each kernel has also a set of thread block size constraints that must be satisfied, determined by its parallel strategies. We support the following constraints, with compatibility guaranteed by code generation:

1. Exact number of threads;
2. Multiple of n; or
3. Power-of-2.

Given the kernel geometry and constraints, we determine the best size for each thread block and extrapolate the number of blocks according to the following formula:

$$\text{Blocks} = \lceil \frac{\text{Size}}{\text{Block Size}} \rceil$$

The number of blocks is thus the quantity required for the complete computation. Note that some threads may therefore be inactive and require bounds checking. We discuss the selection of thread block size for each GPU-capable geometry below.

Vector: Determined by the resource usage (e.g. registers), each kernel has an upper bound on the number of threads per block. For vector geometries we therefore select the maximum thread block size that obeys both the constraints and the kernel limits. Shown in Figure 6.2 are example thread layouts for a vector geometry of size 4096 under each of the above constraints (with no kernel limit).

```
Block Size = 768
Blocks     = 6
```

(a) Exact size = 768.

```
Block Size = 1000
Blocks     = 5
```

(b) Multiple size = 100.

```
Block Size = 1024
Blocks     = 4
```

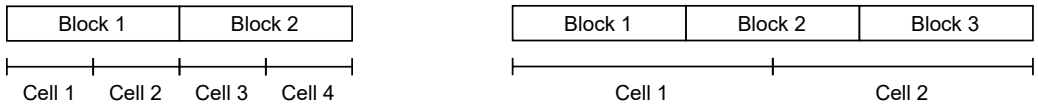
(c) Power-of-2 size.

```
Block Size = 1024
Blocks     = 4
```

(d) *Unconstrained.*

Figure 6.2 – Thread block layouts for vector geometry with runtime size 4096.

List: List kernels specify constraints on the number of threads per cell, rather than the thread block. Each thread block thus uses the maximum size supported by the kernel, and the generated code assigns threads to each cell as needed. Cells may therefore be split unevenly across multiple blocks, or multiple cells assigned to a single block. We show examples of both layouts in in Figure 6.3. If the cell size is unconstrained, we use the average of runtime sizes and execute larger cells using the strategy shown in Chapter 4. The number of thread blocks is given by the total number of threads required to execute all cells.



(a) Multiple cells in each block.

(b) Cells span across multiple blocks.

Figure 6.3 – Thread block layouts for list geometry.

6.2 CUDA Runtime

We build a wrapper for the CUDA runtime [165] and driver [163] environments, providing a unified interface for managing data and executing kernels that require varying high and low-level access. In particular, we implement the following:

- Devices and contexts;
- Events and errors;
- Data buffers and constants;
- Kernels and invocations;
- **Compiler and linker**; and
- **Internal and external libraries**.

Most elements are simple wrappers, with the exception of compilation, linking and libraries described in Sections 6.2.1 and 6.2.2 respectively. We discuss data buffers, allocations and transfers in Section 6.3.

6.2.1 Compiler and Linker

The compiler and linker wrappers are key components in our pipeline, producing a linked CUDA binary from sources of varying formats. Architecture-independent PTX code is either compiled by our own assembler, or `nvPTXCompiler`, a newer NVIDIA library for runtime compilation equivalent to `ptxas` [169]. We use our own assembler for query compilation due to its low cost, and NVIDIA's for offline or cached compilation as the extra overhead is inconsequential. We specify the following compilation options to `nxPTXCompiler`:

- Compute capability for the current device;
- Optimization level (default `-O3`, supports `-O0`/`-O1`/`-O2`);
- Allow expensive optimizations, trading increased compilation time for (possibly) better performance; and
- Compile only, generating a relocatable binary.

The resulting binary is passed to the CUDA linker along with external libraries described below. The fully linked executable is compatible with the CUDA runtime.

6.2.2 Libraries

Complex math functions like `@sin` and `@cos` require extensive optimization and testing beyond the scope of this thesis on SQL queries. We therefore rely on external libraries for their support, namely `libdevice`, a low-level library distributed as part of CUDA [166]. Represented in generic LLVM bytecode [127], we must therefore first generate an architecture-specific binary using the LLVM and CUDA toolchains as done in various GPU projects [150, 184, 218, 3, 4] which were used as guidance.

We use the NVPTX LLVM backend [186], translating from bytecode to PTX and performing necessary `-O3` optimization. Note that the supplied bytecode is missing the *target triple* and *data layout* required by LLVM backends and must therefore be set before compilation [184]. As the complete library is extensive, we extract necessary functions by linking a dummy module shown in Figure 6.4 and keeping only those which match. This reduces the system initialization time and produces a smaller binary. The resulting PTX code is assembled to a relocatable ELF file using NVIDIA's backend compiler and cached during start-up. If required by the query, external libraries can be linked when generating the final executable.

```
declare float @__nv_cosf(float)
declare double @__nv_cos(double)
declare float @__nv_sinf(float)
declare double @__nv_sin(double)
[...]
```

Figure 6.4 – Dummy LLVM module for quickly extracting necessary functions.

We also implement an optimized internal library in PTX for data initialization (e.g. min, max) of various types, nested kernels for the `@like` library, and `@group` dictionary construction. Compiled and linked using NVIDIA's pipeline, we cache the binary on start-up and execute kernels as needed by the runtime. Note that we do not use our own assembler as the compilation occurs during system initialization and not when executing the query. We thus benefit from additional optimizations provided by NVIDIA without any execution overhead. Unlike external libraries, we maintain a separate CUDA binary as the functions are never linked or called by other kernels.

6.3 Data Management

Data buffers store program values of varying types, either on the CPU and/or GPU. Each buffer is associated with its HorseIR type and shape, and if GPU-capable, an associated size buffer. Note that even though certain containers may not be GPU-capable, their nested contents may be. We list each kind of data buffer in Table 6.1.

Table 6.1 – Data buffer properties.

Kind	GPU	Description
Constant	✓	Scalar values (size not stored).
Vector	✓	Array values, stored contiguously.
List	✓	Ordered collection of cells, typically vectors or enumerations.
Enumeration	✗	Foreign keys, storing a values vector, a pointer to the foreign vector, and a cached indexes vector.
Dictionary	✗	Key-value mapping, stored as a vector of keys and one vector per bin.
Table	✗	List of columns, either vectors or enumerations.

As GPU data buffers are more complex, we discuss their allocation in Section 6.3.1, transferring in Section 6.3.2, and optimizations for both.

6.3.1 Buffer Allocation

GPU buffers provide data storage as well as attributes required for thread management. Each buffer thus consists of a data allocation, and a separate size buffer. We chose this separation due to simplicity, as buffers can be nested in higher-level representations (e.g. vectors grouped to form a list). We present each type-specific GPU buffer layout and an optimization strategy to reduce allocation cost.

Vector: Vector data is contiguous, storing an array of values. Each buffer consists of a single data segment and an associated 4-byte size buffer that is used for bounds checking and resizing. We show the vector buffer layout in Figure 6.5.

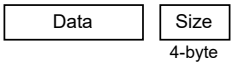


Figure 6.5 – Vector buffer GPU layout.

List: Lists are represented as a vector of pointers, referred to as the *header*, each element pointing to cell data. An identical structure is used for the associated size buffer, with a header pointing to each cell size. We allocate list buffers according to the following schemes, optimized for different applications:

(a) **Cell:** Each cell buffer is allocated separately, as shown in Figure 6.6a. For lists with a small number of cells constructed from independent vectors this proves sufficient, but is ineffective for larger lists typical of grouping.

(b) **Contiguous:** When allocating lists with numerous cells (typical for `@group`), the cost becomes significant. An alternative strategy thus allocates all cells in a single contiguous vector as shown in Figure 6.6b. The header then points to each cell within the allocation as in the previous strategy, and the allocation cost is reduced.

(a) Individual cell-based storage.

(b) Optimized contiguous-cell storage.

Figure 6.6 – List buffer GPU layouts. (Adapted, © 2021 IEEE [125])

The generated code is unchanged regardless of the scheme, as it depends solely on the presence of a header and cells. Note that lists may be formed for independent vectors, requiring cell-based allocation to avoid copying.

Pre-allocation

Repeated buffer allocation is expensive, requiring numerous calls to the CUDA runtime. We reduce this cost significantly by pre-allocating pages before execution and managing the allocations manually using an `sbrk`-style approach. Each allocation increments a pointer, aligned to a multiple of 16-bytes. This guarantees data alignment for all types, including vectors used for string caching. The entire page can also be freed after execution in one step, simplifying cleanup. A similar strategy is used in both BlazingSQL [103] and OmniSci [150], other GPU database systems.

6.3.2 Buffer Transfers

Data buffers are transferred as needed, avoiding excess copying over the PCI-e bus. To do so, we maintain the state of each buffer on the CPU and GPU, similar to the idea behind cache coherence [49]. The runtime then requests a read or write copy for a particular target, and the data is transferred only if necessary. Shown in Figure 6.7 is the equivalent state diagram for our system. We include 4 states:

1. **Empty:** Unallocated buffer.
2. **CPU-only:** Exclusive copy of the data on the CPU.
3. **GPU-only:** Exclusive copy of the data on the GPU.
4. **Shared:** Shared copy of the data, up-to-date on both targets.

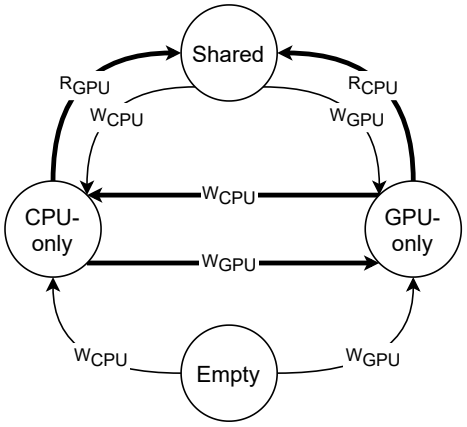


Figure 6.7 – Buffer states and transitions. Bold transitions indicate a data transfer.

Each transition represents the new state when requesting a read or write copy, hiding self-transitions and capturing data transfers in bold. If the target has not yet been initialized, a new buffer is allocated before transfer. We note that reading requires either a shared or exclusive copy, while writing requires an exclusive copy. Transitioning out of an exclusive state triggers a transfer, as the data on the other target is stale. Note that requesting a read copy of an empty buffer is invalid as there is no data and indicates an error in the runtime system.

Pinned Memory

Data transfers between the CPU and GPU are costly, especially for the large buffers required by input tables. This is heightened by CPU memory allocations, which are typically page-swappable and unable to be directly copied to the GPU without an intermediate copy. We therefore use *pinned memory*, allowing the system to transfer data directly over the PCIe bus [87]. As the primary data type, vectors are allocated on pinned pages using a custom allocator[1], optimizing their transfer.

6.4 Summary

The runtime system executes the query and returns the result to the user, invoking the output of earlier compilation phases. CPU control code is interpreted directly at low cost, while fully assembled parallel kernels are offloaded to the GPU for acceleration. As the primary runtime overheads, we optimize data transfers by copying data only when necessary, and minimize the allocation cost by combining buffers into contiguous structures and pre-allocating pages.

[1] https://en.cppreference.com/w/cpp/named_req/Allocator

Chapter 7
Evaluation

In this chapter we present an evaluation of our system, analyzing the impact of each phase and comparing against the current state-of-the-art. Our aim is a compiled query GPU database that operates without the usual compilation overhead for competitive end-to-end performance on short queries. We begin by describing our methodology and comparison systems in Section 7.1, followed by the motivation for our approach in Section 7.2. Next, we show compilation and execution breakdowns for each query in Section 7.3. We then compare our performance against other modern, open-source GPU and CPU database systems on multiple use cases in Section 7.4. Lastly, we evaluate our optimizations for execution and compilation in Section 7.5 and summarize our results in Section 7.6.

7.1 Methodology

We collect results on a single machine described below, averaging each value over 10 consecutive runs and dropping the highest and lowest timings. To account for initialization costs (e.g. cache, filesystem), 5 warm-up iterations are discarded beforehand.

- **OS:** Ubuntu 20.04 LTS (docker)
- **CPU/RAM:** Intel i7-8700K @ 3.7GHz, 32GB DDR4

- **GPUs:** NVIDIA GeForce GTX 1080 Ti, NVIDIA GeForce RTX 3080
- **CUDA:** 11.4.3 and graphics driver 470.94

To demonstrate portability of our approach across platforms, we experiment and show results for both GPU targets. Note that despite multiple PCI-e 16x slots being present on the motherboard, our CPU supports only 16 lanes[1]. We therefore swap the GPUs between experiments as sharing bandwidth greatly degrades performance. Additionally, we use `nvidia-persistenced` [159] to keep the GPU and driver initialized, as we previously experienced crashes when collecting data over long periods[2].

Comparison Systems

We select the latest version of 4 recent and open-source comparison systems shown in Table 7.1. This set of comparison points covers a variety of execution strategies, compiled and interpreted on both the CPU (single threaded) and GPU, allowing us to evaluate the effectiveness of our low-overhead approach. Note that BlazingSQL is built on an open-source library, RAPIDS.AI, which provides parallel implementations of key functions. For rNdN, we also differentiate between our complete pipeline and that which uses NVIDIA's `ptxas` backend in lieu of our simplified assembler. To account for the impact of optimization, we measure at both `-O0` and `-O3`.

Table 7.1 – Comparison systems.

System	Version	Device	Execution
rNdN	Jan 2022	GPU	Compiled
rNdN + `ptxas (-O0/-O3)`	Jan 2022	GPU	Compiled
OmniSci [150]	5.10.0 (Jan 2022)	GPU	Compiled
BlazingSQL [103]	21.08.02 (July 2021)	GPU	RAPIDS.AI
MonetDB [101]	11.41.13 (July 2021)	CPU	Interpreted
HorsePower [38]	August 2020	CPU	Compiled

[1] https://ark.intel.com/content/www/us/en/ark/products/126686/intel-core-i78700-processor-12m-cache-up-to-4-60-ghz.html

[2] https://forums.developer.nvidia.com/t/nvrm-rminitadapter-failed/125897

TPC-H Benchmark

We evaluate our approach on the TPC-H benchmark suite, a collection of 22 queries designed for performance evaluation of relational database systems [221] and frequently used in research. Each query tests a variety of SQL operators for a business context varying from simple to complex. We use scale factor 1, comprising 1 GB combined input data across all tables, as it fits within GPU device memory yet is still significant. Most systems support the entire benchmark suite, with the exception of OmniSci (q14, q20, q21, q22) and BlazingSQL (q11, q15, q21, q22). In addition, we modify some queries for BlazingSQL as they are unsupported by the parsing engine[3].

7.2 Motivation

Before presenting the complete rNdN system, we motivate our focus on minimizing compile time by considering the performance under NVIDIA's optimizing assembler (ptxas `-O3`). Shown in Figure 7.1 are high-level execution and compilation breakdowns for both architectures. Execution time is divided into input data caching (e.g. TPC-H and string pad), intermediate data management (e.g. transfer, initialization, resizing), CPU execution, GPU kernels and runtime analyses, and overhead. Compilation is partitioned into compiler and assembler, corresponding to phase 1 and 2 of the pipeline, with binary loading included in the latter. We therefore evaluate the execution strategies described in Chapter 4 (Frontend and Compiler) and Chapter 6 (Runtime), paired with the proprietary assembler. Times are presented in milliseconds, with the compilation scale 3x that of execution due to its high cost. Bars which exceed the plotted region are annotated with their value.

Measured on the TPC-H benchmark, compilation dwarfs execution in all queries (n.b. scale difference), and is the largest and limiting factor for end-to-end performance. This is especially relevant for short-running queries as any speedup from compilation is offset by its overhead. Moreover, the compilation cost is nearly entirely in the second phase that generates the CUDA binary from intermediate PTX

[3] https://github.com/BlazingDB/blazingsql/pull/785

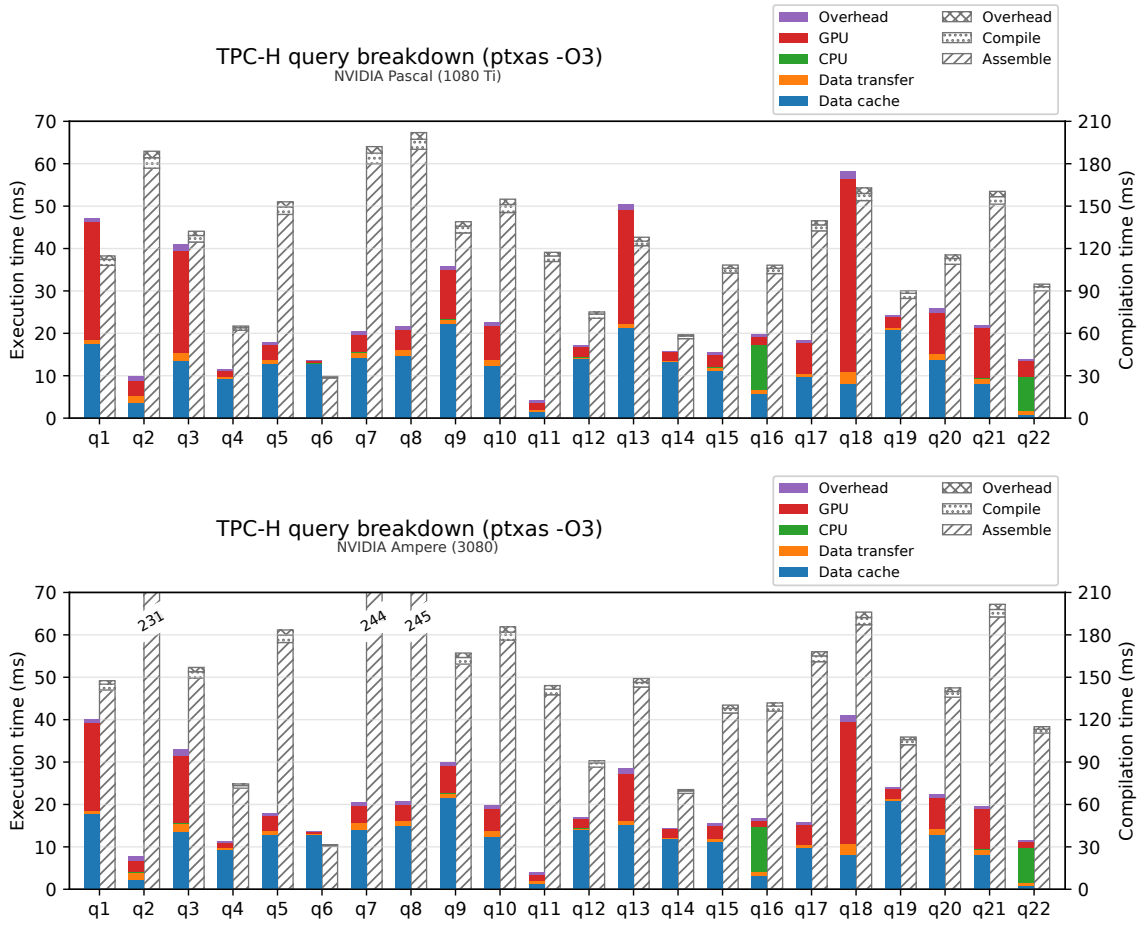


Figure 7.1 – rNdN execution and compilation breakdown with ptxas `-O3` backend for both Pascal (1080 Ti) and Ampere (3080) architectures. Execution is shown in the left bar (left scale) and compilation in the right bar (right scale; 3x execution).

code. It is thus a crucial bottleneck, though further investigation is extremely limited due to its closed-source implementation. This formed the basis of our approach, developing a runtime-suitable replacement assembler that exploits a greatly simplified pipeline without excessively slowing execution. Interestingly, the compilation cost is higher on Ampere, possibly due to target-specific optimizations or code generation strategies. On the other hand, GPU execution time is markedly lower. Note that data transfers are primarily limited by the PCI-e bus and are therefore unchanged.

As an obvious solution to high compilation times, we also evaluated the performance of the proprietary backend without optimizations (`-O0`, allow expensive optimizations `false`[4]). Shown in Figure 7.2 is the equivalent execution breakdown under this scheme using the same split scale (compilation 3x execution). Unfortunately, despite a substantial decrease in compilation time, execution time has been severely degraded on both architectures. Optimization is thus essential to high performance using NVIDIA's assembler, although it comes at a high cost.

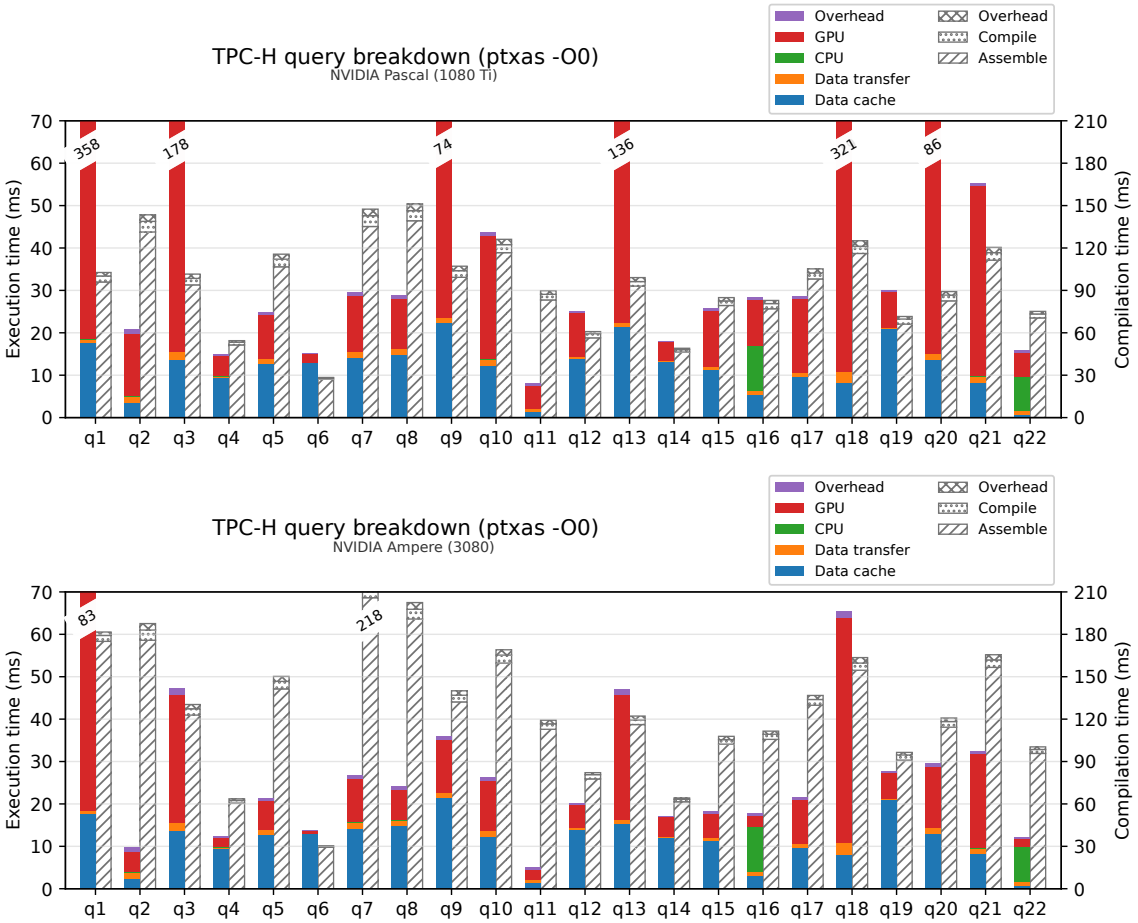


Figure 7.2 – rNdN execution and compilation breakdown with ptxas -O0 backend for both Pascal (1080 Ti) and Ampere (3080) architectures. Execution is shown in the left bar (left scale) and compilation in the right bar (right scale; 3x execution).

[4]Preliminary experiments showed no significant impact from this flag.

7.3 Execution Breakdown

We next present the equivalent execution and compilation breakdowns for our complete end-to-end rNdN system, shown in Figure 7.3, measuring the time spent in each phase for both architectures. As in the motivation, execution is divided into input data caching, intermediate data management, CPU execution, GPU kernels and runtime analyses, and overhead. Compilation is partitioned into compiler and assembler. Times are presented in milliseconds, with both bars using the *same* scale. A further decomposition of kernels and compilation are presented in Sections 7.3.1 and 7.3.2.

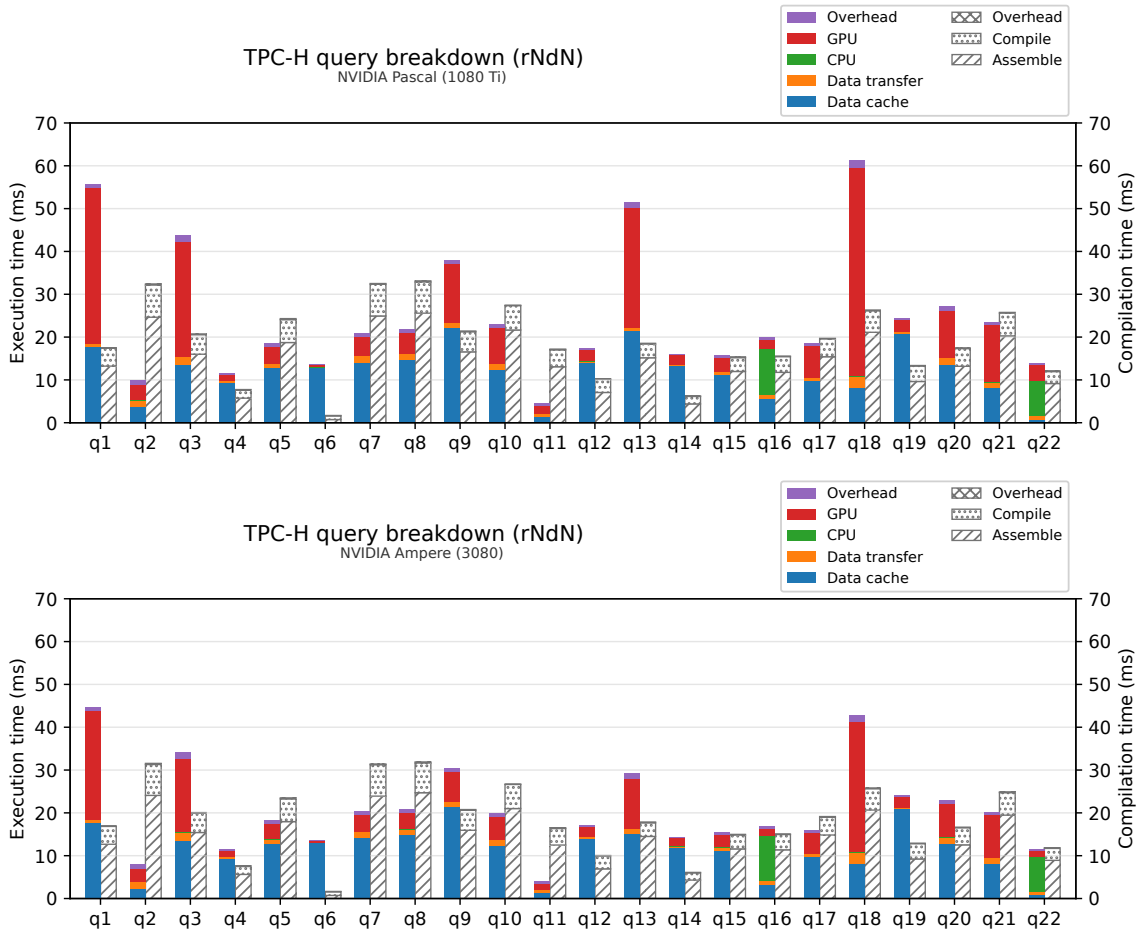


Figure 7.3 – rNdN execution (left bar) and compilation (right bar) breakdown for both Pascal (1080 Ti) and Ampere (3080) architectures.

Overall, query execution is dominated by data caching, with TPC-H input tables representing the primary cost for many queries. This confirms prior work that data costs must be considered for GPU benchmarking [84], and demonstrates the importance of effective caching and transfer strategies. Intermediate data transfers and management (e.g. resizing and initialization) are insignificant in comparison. In terms of computation, most of the execution is GPU-based, with exception of string operations `@substring` (q22) and `@order` (q16). In both cases, we assessed the cost of implementation too high for their limited use, although optimization is possible in future work. Our code generation approaches are thus sufficient to parallelize a wide variety of queries. The remaining execution time is deemed overhead (e.g. interpretation) and negligible in our evaluation. While examined in more detail in the following section, we note that queries with high GPU-execution times (q1, q3, q18) typically group large amounts of data. In these cases input data is relatively less costly, though we believe that further optimization of the library function is possible and would again show the importance of data management.

Compilation time is still heavily skewed towards the backend compiler (assembler), with the frontend significantly faster. This is intuitively due to the simplicity of the frontend representation (HorseIR), which utilizes high-level constructs for complex operations and contains no explicit control-flow. Data-flow analyses that typically dominate compilation are thus simplified and more efficient. In comparison, the assembler operates on a quasi machine-level IR (PTX) where complex operations are decomposed into sequences of basic instructions and typically contain branching and loops. Expensive fixed-point analyses are thus required; the effect of which is especially pronounced in queries with larger programs (q2, q7, q8, q10). By comparison, short and simple queries (q6, q14) are much faster to compile both in the frontend and backend. Compilation time thus remains an important factor in query performance, and while it is usually less costly than execution, still merits significant optimization. This is especially true for the phase 2 assembler pipeline.

Comparing both architectures, we see an identical trend throughout the entire benchmark, with improvements in Ampere over Pascal derived from faster GPU execution. Other execution costs remain essentially unchanged as the host system was

identical. In particular, data transfers are likely limited by the PCIe bandwidth – a host property – making improvements in GPU design inconsequential for transfers. As kernel performance continue to improve, data management therefore becomes increasingly important and requires newer PCIe versions to optimize transfers. Compile time is likewise unchanged, as we use equivalent strategies for both systems.

Our system is also well-suited for runtime use, demonstrating execution comparable to ptxas `-O3` with compilation time significantly lower than `-O0`. We thus match or exceed the best case of each measure. Additionally, we note that execution is only marginally worse on Ampere than Pascal relative to NVIDIA's optimizing compiler. This hints at low payoff in ptxas, as their Ampere compilation pipeline is significantly slower than Pascal while ours remains constant. We do, however, follow a similar trend in compilation cost, with complex queries taking longer to compile and the second phase remaining the most expensive.

7.3.1 GPU Execution

We further breakdown the execution time on the GPU, showing the proportion of time spent in the main outlined kernels and library functions in Figure 7.4. In nearly all cases, library functions represent a significant part of query execution – in only a few exceptions (q4, q6, q10, q12, q14, q15, q19) do outlined kernels represent the majority. This is unsurprising, as most non-library functions are simple-to-execute element-wise unary and binary operations, and their intermediate data cost is eliminated through outlining (see Section 7.5.2). Library functions are therefore important optimization targets, with particular emphasis on `@group` and `@join_index`. While sorting may seem a significant cost for some queries, its high proportion comes from the sort required for grouping. Sorting for non-grouping purposes on the GPU is negligible and is typically used to order output data (always in our benchmarks). Joins vary in cost depending on the query, with some more expensive than others, although remaining an important optimization target. Other library operations are for the most part negligible, with the exception of `@like` in q13 and `@member` in q22. With respect to architecture differences, a similar trend is present throughout the benchmark. We

evaluate the impact of library optimizations in Section 7.5.1.

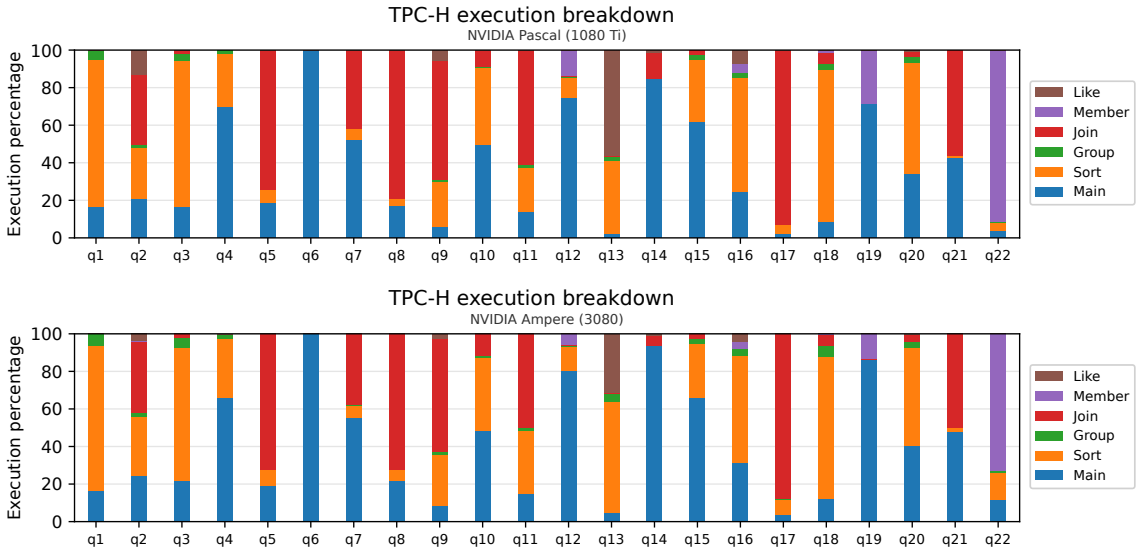


Figure 7.4 – rNdN GPU kernel execution breakdown.

7.3.2 Compilation

Lastly, we analyze the compilation time, describing the proportion of each phase in our pipeline. Presented in Figure 7.5 are the compiler and a detailed view of the assembler. We include the frontend, outliner, and PTX code generation for the first phase, and control-flow structuring, register allocation, SASS code generation, instruction scheduling, peephole optimization and binary generation for the backend. Overhead is excluded as it is a negligible part of compilation; CUDA binary loading and assembly are included in binary generation.

As shown in the overview, translating from HorseIR to PTX represents only a small part of the compilation time for nearly all queries, owing to the efficient representation. Code generation is the primary cost, followed by outlining and then the negligible frontend. A notable exception, q6, spends a considerable proportion of time in the frontend compiler as it is relatively simplistic. In all other queries, the backend compiler phases are much more expensive due to the large amount of

PTX code generated from HorseIR. Additionally, fixed-point analyses are used for structuring (dominators and post-dominators) and register allocation (live variables). Register allocation is in fact the most costly part of the backend pipeline proportionally speaking (the absolute value is small). On the other hand, scheduling requires only a basic dependency analysis as it operates at the basic block level, though it remains costly. Together, these backend phases represent a significant part of the overall pipeline and were extensively optimized for runtime use. This highlights the importance of our design choice, selecting fast compilation techniques for expensive operations over those with more precise results. Peephole optimization and binary generation are both extremely fast, as is SASS code generation.

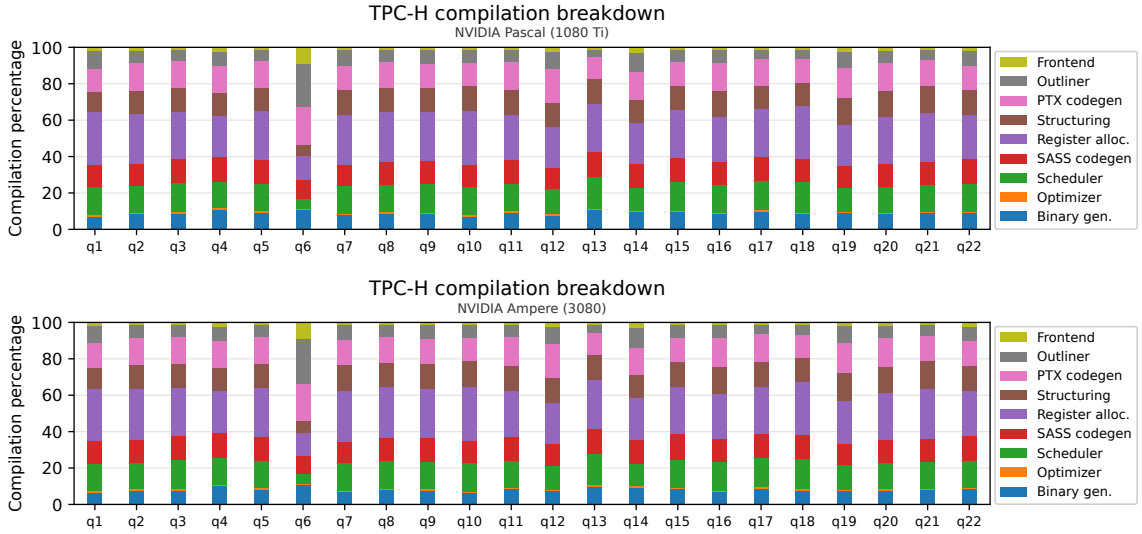


Figure 7.5 – rNdN compilation breakdown.

7.4 Performance Comparison

We evaluate our performance against the comparison systems from Section 7.1 using 4 metrics, corresponding to different use-cases. Our selected metrics include:

1. **Compilation Time:** Compilation time before execution, measuring the added cost of query compilation (Section 7.4.1);

2. **Cached Execution:** Pre-compiled queries and GPU-resident data, for repeatedly executed analytics queries with data already loaded (Section 7.4.2);

3. **Uncached Execution:** Pre-compiled queries but data stored on the CPU, considering applications with fresh data (Section 7.4.3); and

4. **Total Execution:** End-to-end performance, a key comparison point for evaluating trade-offs between execution engines and targets (Section 7.4.4).

Performance is measured by speedup, with values greater than 1 showing better results on our system and values less than 1 demonstrating slowdown. If a comparison system does not support a query, it is indicated by 'x'. Note that we use a non-linear speedup axis to highlight the important range and extract meaningful data.

7.4.1 Compilation Time

As query compilation is central to our approach, we begin by analyzing compile time performance compared to other compiled databases: HorseIR on the CPU and OmniSci on the GPU. In addition, we include rNdN with NVIDIA's backend at optimization levels `-O0` and `-O3`, measuring the impact of our runtime-optimized assembler. For OmniSci, note that hash table building for joins is performed partly at compile time (referred to as "reify"). We therefore subtract `buildJoinLoops` from the compilation time for fairness of comparison, and later include it with uncached execution. Due to its integration with code generation, this is imperfect but more representative. For HorsePower, we use the `gcc` compilation time once the C program is generated, excluding translation from HorseIR as the built-in profiling is limited. Our system measures the compilation cost from input HorseIR program to assembled and loaded CUDA binary. Shown in Figure 7.6 is the speedup graph for the compilation time for the entire benchmark. Encouragingly, we see significant performance improvement compared to all other systems and geomeans exceeding **5x** on both platforms.

Compared to NVIDIA's backend compiler, we see significant performance improvement both with optimizations enabled and disabled, despite the expected quality difference of generated code. This hints that their infrastructure requires extensive

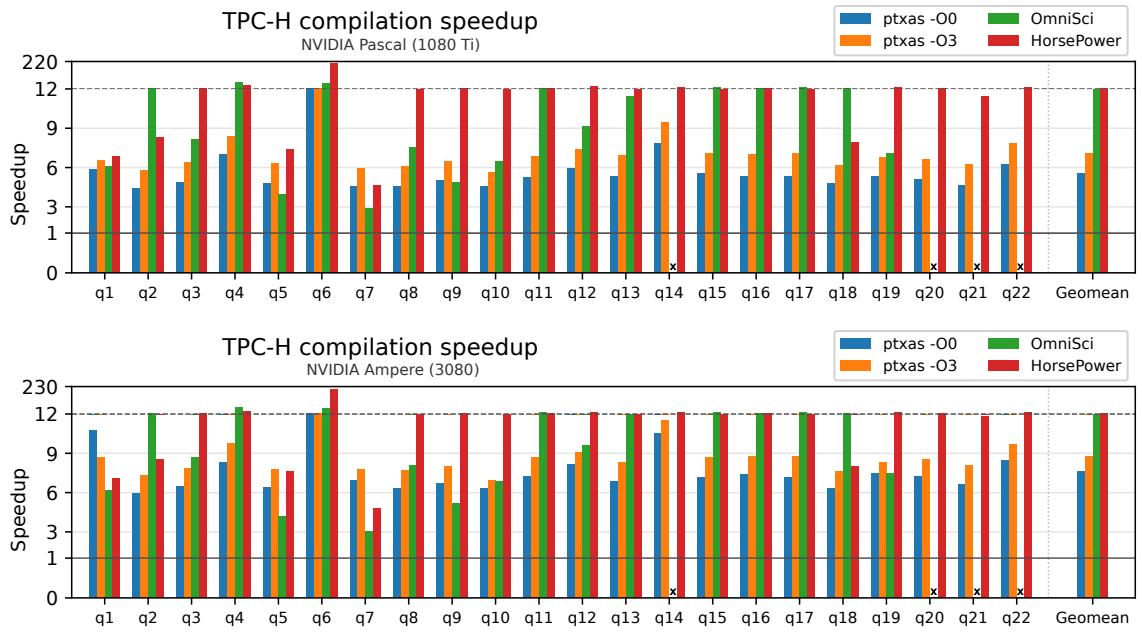


Figure 7.6 – **Compilation** speedup of rNdN (compiled systems only). Note the non-linear axis below 1 and above 12 that highlights the important region.

work to translate code regardless of optimization level. In fact, compiling an empty module with no code still has considerable cost, likely due to initialization overhead or binary generation. In addition, as noted in the execution breakdown, their compilation on the newer architecture is significantly more expensive while we maintain parity. Our approach is thus much more suitable for runtime systems than the proprietary pipeline, provided that our execution is sufficiently strong. We evaluate the impact on execution in Section 7.4.2 and the overall trade-off in Section 7.4.4.

OmniSci utilizes the LLVM compiler framework for its compilation pipeline [215]. They first generate LLVM IR, translate to PTX, and assemble a CUDA binary using the GPU driver API – the same kind of backend used in the ptxas comparison points. We therefore see significant speedups in all queries, as their system is limited by the same slow backend compiler provided by NVIDIA. Their compilation time also exceeds our system with NVIDIA's backend in nearly all cases, due to the added cost of the LLVM compiler infrastructure. Perhaps unsurprisingly, as HorsePower employs

`gcc`, an ahead-of-time compiler, we see similar performance improvements. Note that q6 is very simple and gives an extreme speedup that we do not expect in most cases.

7.4.2 Cached Execution

Cached execution measures the query performance excluding compilation and input data transfers, allowing for all types of caching (including pre-computation of hash tables in OmniSci). It therefore represents cases where a query is repeatedly executed on unchanging data that fits in GPU memory, isolating our algorithmic and runtime design. Shown in Figure 7.7 are the speedups in relation to all comparison systems from Table 7.1, both on the CPU and GPU, compiled and interpreted.

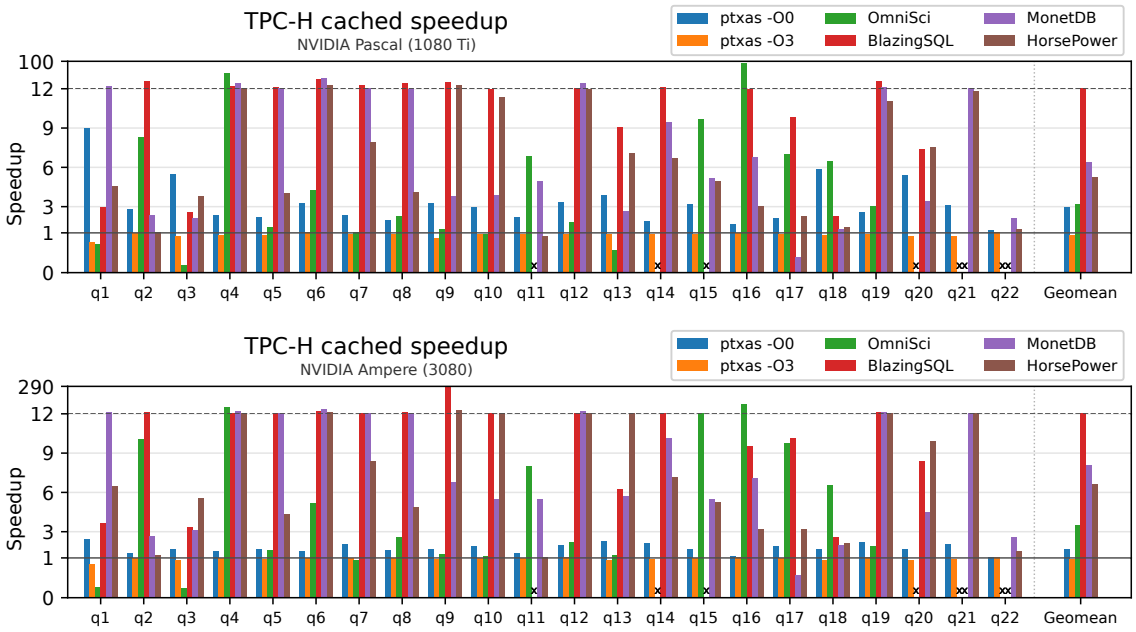


Figure 7.7 – **Cached** speedup of rNdN, excludes input transfer and compilation.

With the exception of NVIDIA's optimizing backend, we see performance improvements on a variety of queries with geometric mean speedup. Most importantly, we outperform ptxas baseline with `-O0` and only see small degradation (<**6%** geomean) compared to the optimizing `-O3` variant. This indicates that our simplistic approach

is effective at extracting high performance from queries, especially considering the significantly improved compilation time. This emphasizes the novelty of our design, demonstrating that only basic optimizations are required for fast execution of SQL queries on the GPU. Moreover, our design is portable across architectures, with only a small geomean reduction in performance compared to NVIDIA's optimizing backend on both GPUs (Ampere: **0.97x**; Pascal: **0.95x**). Interestingly, their non-optimizing backend produces higher performance code for Ampere than Pascal. The performance change in the optimizing backend is much less pronounced between architectures.

For CPU systems MonetDB and HorsePower, we show speedup in nearly all cases, with notable geometric means. This holds true on both architectures despite Pascal not being as powerful. The most obvious speedup exception, q17, requires additional investigation of our join library implementation. Compared to GPU systems OmniSci and BlazingSQL, we see significant performance improvements and geometric mean speedup. BlazingSQL has slower execution in all queries, and the exceptions with OmniSci are typically due to our grouping implementation (q1, q3). As previously discussed, we implement group by sorting input data and finding unique keys. This is effective for small data, but requires a more efficient approach as data sizes grow. Encouragingly, we are able to support the entire benchmark suite, whereas the more mature GPU databases are both incomplete. While we are not the first GPU implementation to achieve this feat, it is nevertheless challenging.

7.4.3 Uncached Execution

Uncached execution corresponds to scenarios where input data is not yet stored on the GPU but the query itself is unchanging and pre-compiled. GPU database systems thus have additional cost compared to those based on the CPU, as they need to transfer input tables over the PCIe bus. As shown in Section 7.3, this represents a significant overhead for most queries. Note that due to limitations of BlazingSQL logging and control over transfers and its unrealistically high cost, we exclude data caching from its evaluation and only include the computation. For OmniSci and rNdN, each query is evaluated on an uninitialized system with an empty data cache

(database restarted between iterations) and the compilation time excluded. As mentioned previously, we include the cost of building join hash tables for OmniSci as it represents data computation rather than compilation. Shown in Figure 7.8 are the uncached speedups of our system on the complete TPC-H benchmark suite.

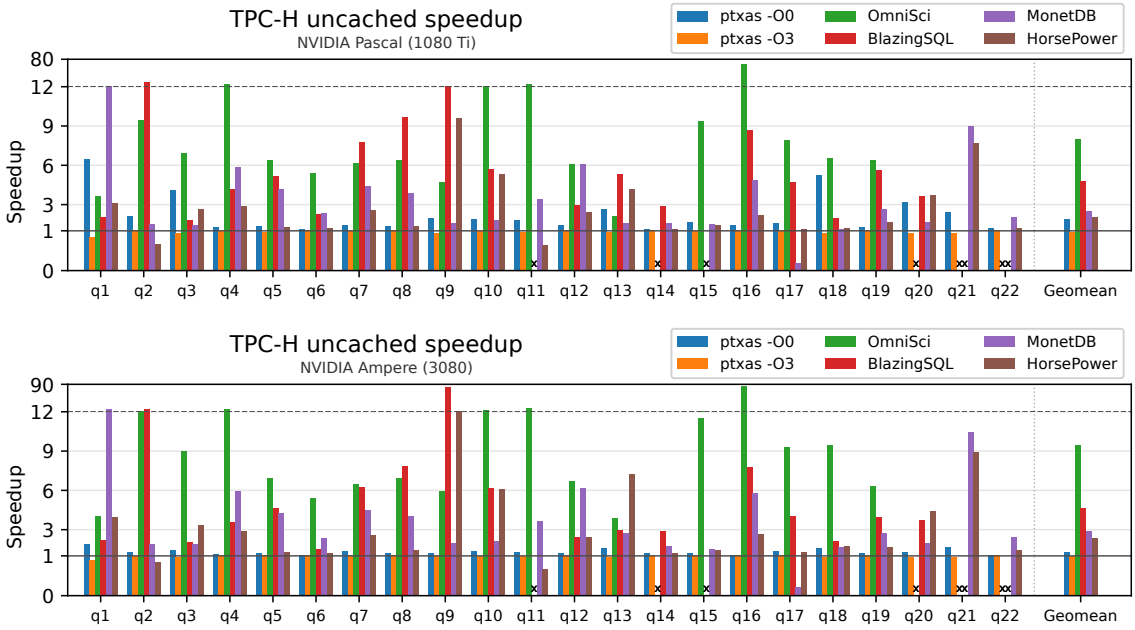


Figure 7.8 – **Uncached** speedup of rNdN, excludes compilation.

Overall, we maintain geometric mean speedups over all comparison systems in both architectures with exception of NVIDIA's optimizing assembler. However, as the data transfer cost for rNdN is independent of the kernels, the gap is smaller than in the cached evaluation. For other GPU database systems, the performance depends on the data management strategy. OmniSci shows substantial slowdown due to its design for clusters and large data sizes, whereas we target single node machines and do not consider partitioning. Interestingly, we maintain speedup on all queries compared to BlazingSQL despite excluding their (extremely costly) input data transfers. Compared to CPU databases, our performance has substantially decreased in relation to cached results due to the cost of data caching. Efficient buffer management is thus essential for highest performance, although we maintain geometric mean speedups

compared to both HorseIR and MonetDB. Data management is especially important for multiple GPU systems and as the data scales.

7.4.4 Total Execution

Lastly, we evaluate the end-to-end performance, including both compilation and data caching. This represents cold-start scenarios, or those where the query and data are frequently changing. It thus allows us to analyze the cost-benefit of query compilation over interpretation for both GPU and CPU systems. We also evaluate the key property of our assembler, trading between compilation and execution for competitive end-to-end performance. Shown in Figure 7.9 are the total speedups for both GPUs.

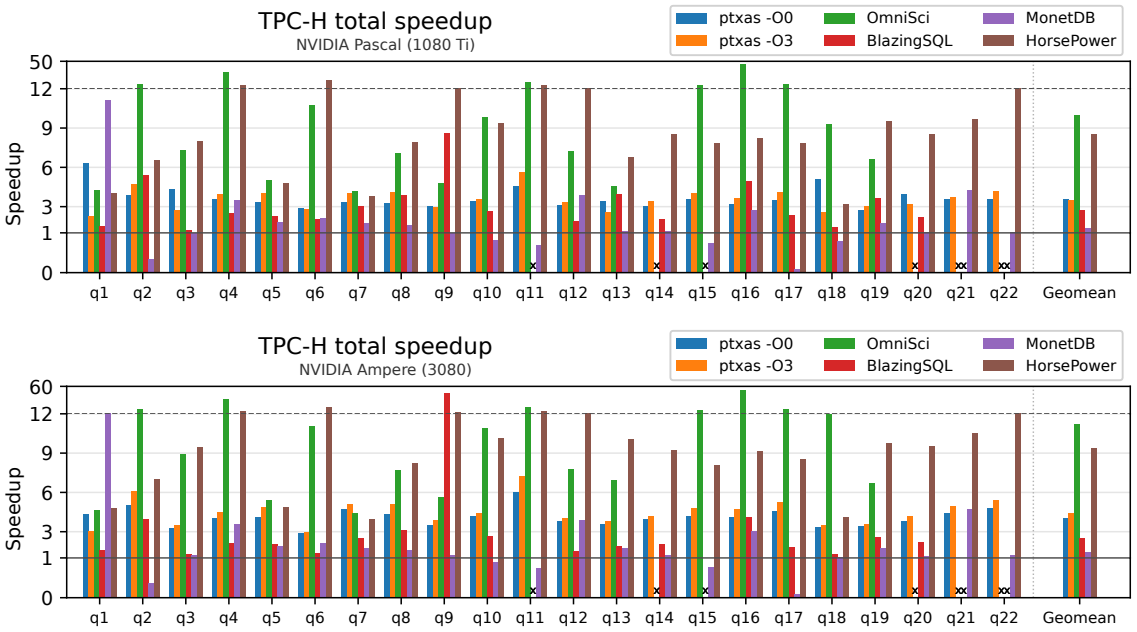


Figure 7.9 – **Total** speedup of rNdN, representing a cold-start.

Compiled database systems typically use compilers designed primarily for ahead-of-time use, as discussed in the compilation comparison. We thus see significant speedups compared to both OmniSci and HorsePower, even for queries that may execute slower in our implementation. A runtime-suitable compiler is thus essential for

end-to-end performance when compiling database queries. With respect to NVIDIA's backend compiler, our approach shows improvement across the board, exploiting a simplified compilation pipeline that still generates high performance code. This remains true for both optimization levels, demonstrating high payoff from rNdN's limited optimizations and success for our trade-off strategy. We also see comparable results for both architectures, showing the portability of our approach.

Compared to interpreted systems, we still see overall performance increases, although unsurprisingly much less than cached scenarios due to the compilation cost. Note that this is a *reversal* from our initial system design which used NVIDIA's backend for assembly. In the prior design, while we achieved high cached performance, our system was unable to compete on end-to-end performance. Indeed, we show significantly higher relative speedup over MonetDB and BlazingSQL than do either ptxas results (which show slowdown). Of the exceptions where MonetDB performs better than our approach, we observe either that compilation greatly outweighs execution (q2, q11), our cached performance was already worse (q17), or the combined overhead with data and compilation was too large. Additionally, both other compiled databases show slowdown compared to interpreted systems, owing to their compilation strategies. Interestingly, a BlazingSQL developer has stated that interpretation was preferred over compilation in their implementation due to cost, instead using an optimized library[5]. Our work challenges this notion, and demonstrates that a runtime-targeted compiler for GPU databases is effective at offsetting compilation overhead and data transfers with improvements in execution. Their application can thus extend beyond cached use cases and into end-to-end scenarios.

7.5 Optimization

Our approach exploits key optimizations for performance, split between execution and compilation strategies. We discuss library and data optimizations in Section 7.5.1 and compilation and assembly techniques in Section 7.5.2.

[5] https://news.ycombinator.com/item?id=19197133

7.5.1 Execution

Library functions: sort, join, group

As discussed in Section 7.3.1, library functions represent a significant portion of execution for most queries with our implementation strategies having great impact on performance. Shown in Figure 7.10 are the speedups of 3 optimized library functions, sort, group and join, over our naive baseline implementations. We measure the impact on cached query performance and not the isolated operation as it better captures their influence. Higher bars indicate better performance for our optimized implementation, while queries that do not utilize the algorithm are noted with 'x'.

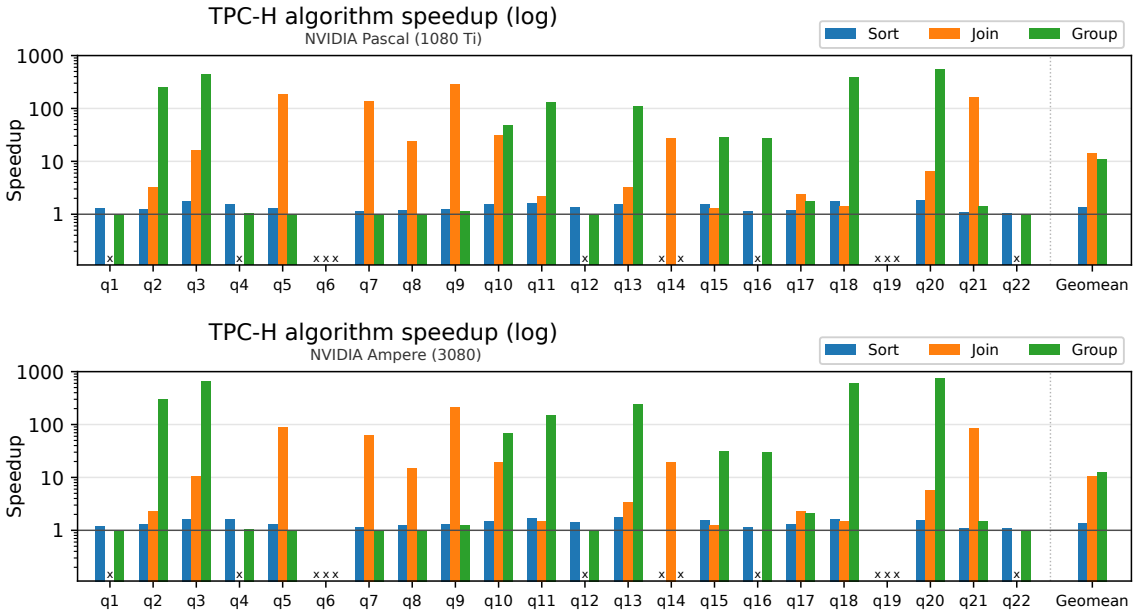


Figure 7.10 – Query speedup from algorithmic and data layout optimizations.

Beginning with sort, we implement two variations of in-place bitonic sort: an optimized implementation with shared memory and a naive device memory-only approach. Recall that as an iterative algorithm, data is sorted into sequences of increasing length. We can therefore cache data in shared memory for some steps, reducing the use of slower device memory. Results show a modest improvement, mostly due to its use in grouping, although we found that power-of-2 bitwise operations discussed

in Section 4.3.5 were more impactful. Sorting for non-group purposes is already sufficiently quick and thus optimization has little impact on the overall query.

Furthering our sort optimization, we implement an efficient buffer allocation strategy for group output. Once the data is sorted, a key-value store (dictionary) is constructed, with each key pointing to a vector of data indexes representing elements in the group. For dictionaries with a small number of keys we can individually instantiate vectors, but this is not scalable to larger results. We therefore implement a buffer optimization, storing data for all groups in a single contiguous (compressed) vector as discussed in Section 4.3.4. This yields substantial improvements for dictionaries with numerous keys (q3, q18), but as expected has little impact on smaller results (q1). Further optimization is required to compete with other GPU database systems on all queries, likely moving to a non-sorting approach [114], but is left as future work.

Joining is well known as an expensive operation with significant impact on performance. Our initial implementation was simplistic, computing the result through a general approach equivalent to nested loops. While supporting all join predicates, it proved much too computationally expensive for good performance, even with extensive optimization. We therefore implemented an optimized hash join for any join predicate which contained at least one equality, with fallback only if necessary. Performance unsurprisingly increased across the entire set of queries, yielding competitive performance with other comparison systems. This holds even for imperfect hashing cases (q17, q20, q21) which only hash on equality attributes. No fallback was present in the current benchmark, and so such cases are left as future work.

Library functions: like

Strings are represented as offsets into a larger structure (dictionary encoding), efficiently supporting equality on the GPU without the need for string contents. While this is sufficient for most queries, those that evaluate restricted regular expressions (notably wildcards) using `@like` are limited in performance. We thus provide an optimized strategy that selectively transfers string data and parallelizes evaluation of the pattern. Importantly, this *caching kernel* allows us to circumvent bandwidth

limitations of the PCI-e bus while maintaining hash values used for fast equality. For comparison, we evaluate the performance against a pure CPU implementation, and an unoptimized GPU strategy that transfers the string pad for the *entire* database rather than a subset (referred to as "uncached"). Shown in Figure 7.11 are the cached execution speedups of our fully optimized approach over the alternatives when including the string data transfer cost (a slight difference to the previously discussed cached scenario). Note that this is not a perfect comparison as queries evaluate `@like` on a subset of TPC-H input data that must also be transferred to the GPU. We chose this comparison over a fully uncached interpretation to isolate the impact of string data.

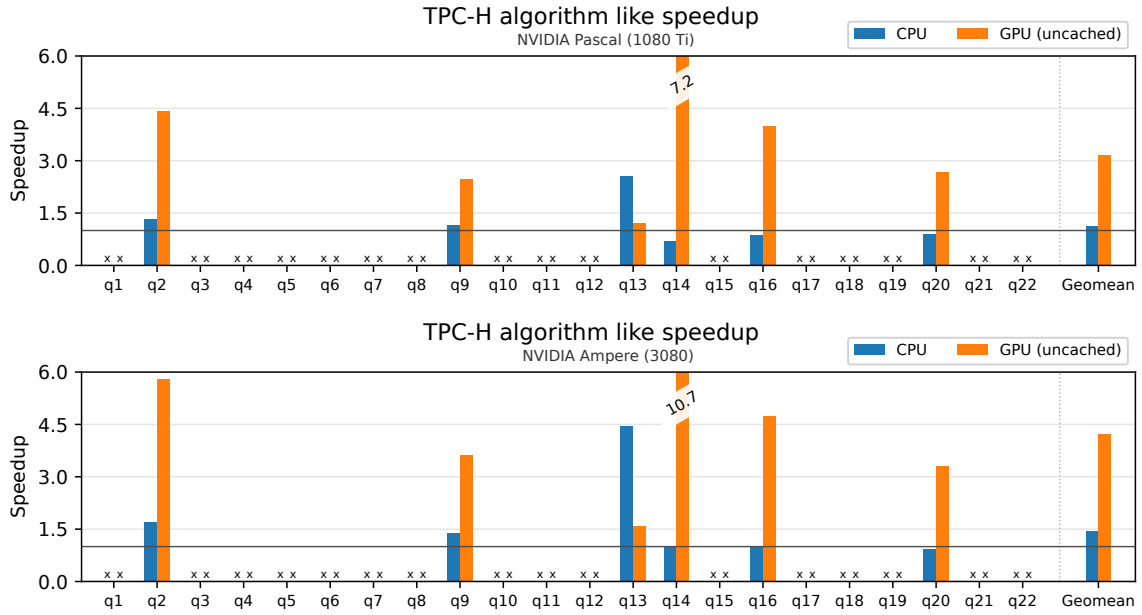


Figure 7.11 – Cached query speedup from `@like` optimization over CPU and naive GPU approaches. Only queries which execute `@like` are measured.

Selective caching improves performance over transferring the entire string pad in all cases, effectively exploiting the PCI-e bus bandwidth. In fact, the cost of transferring unfiltered string data mostly outweighs the benefits – GPU uncached performs worse than a pure CPU approach in all but one query. Compared to the

CPU implementation, selective caching achieves geomean speedup, though individual performance depends both on the data size and the complexity of the pattern. Some queries show significant improvement and others slight degradation. Listed in Table 7.2 are the patterns for each query, with all queries except q13 and one `@like` of q16 operating on string data in the `part` table.

Table 7.2 – `@like` match patterns for TPC-H queries (`%` is a wildcard).

Query	Pattern	TPC-H Table
2	`%BRASS`	`part`
9	`%green%`	`part`
13	`%special%requests%`	`orders`
14	`PROMO%`	`part`
16	`%Customer%Complaints%`	`supplier`
	`MEDIUM POLISHED%`	`part`
20	`forest%`	`part`

In terms of performance, we observe that the CPU implementation is effective for cases ending with a wildcard, as matching begins from the first character. The increased parallelism on the GPU tends to help cases that begin with a wildcard, as matching is more expensive and we can offset the transfer cost. We also note that q13 is efficient on the GPU as operates on a large table and its pattern begins, ends, and contains an intermediate wildcard. Heuristics for selecting the execution engine based on the nature of the matching expression are an interesting future direction.

Data Allocation

Lastly, we evaluate the impact of our GPU buffer allocation strategy described in Section 6.3.1, using pre-allocated pages instead of calling CUDA for each individual buffer. Shown in Figure 7.12 are the uncached query speedups including the cost of input data allocation. As expected, using a simple pre-allocation strategy with little overhead greatly increases performance for both Pascal and Ampere. Although not a direct relationship, queries with fewer allocations tend to benefit less (q6) than those with more (q2). However, this relationship is more complex as there are numerous exceptions, and also relates to the sizes of each allocation. We observed in preliminary

experiments that larger data took correspondingly longer to allocate in CUDA, likely pointing to initialization cost or time to locate a suitable segment. As our pages are pre-allocated and we increment a simple pointer for each new buffer, we avoid any relationship with size and leave initialization to the runtime system as required. Comparing between architectures, we see no significant difference in cost and a similar trend throughout.

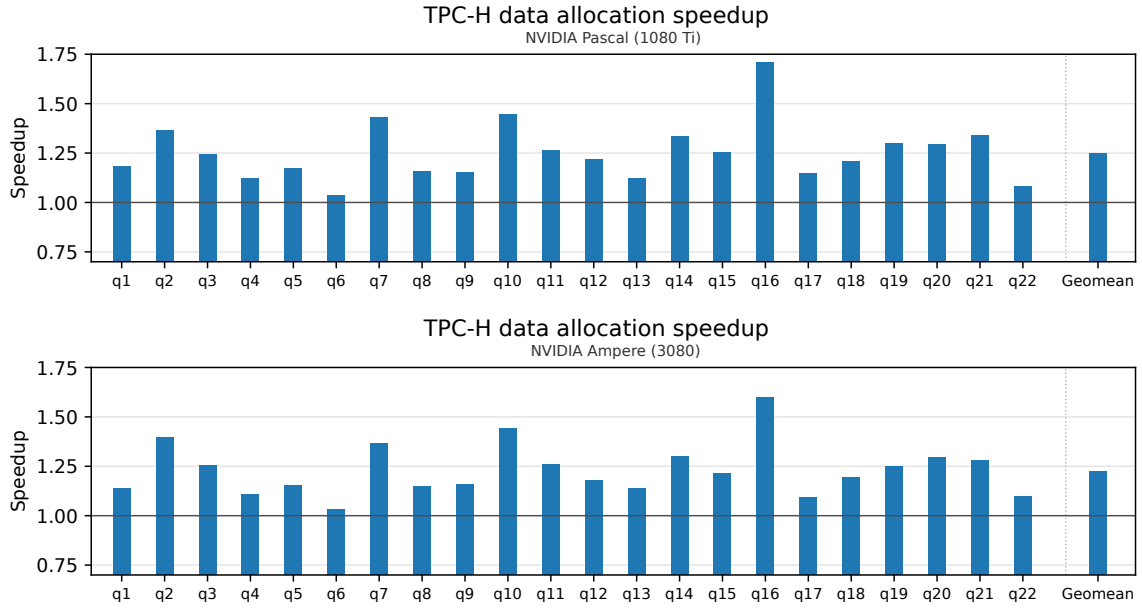


Figure 7.12 – Query speedup from data allocation optimization.

7.5.2 Compiler and Assembler

Outliner

Outlining is divided into two phases, a first pass that merges data-dependent kernels of identical geometry, and a second pass that allows for compressed geometries and shared input data. Shown in Figure 7.13 are the speedups for each query, evaluating the performance over a naive baseline that uses a distinct kernel for each operation. We evaluate the impact of the first pass in isolation (Flow), the complete optimizing outliner (Full), and the relative speedup from the second pass (Flow/Full).

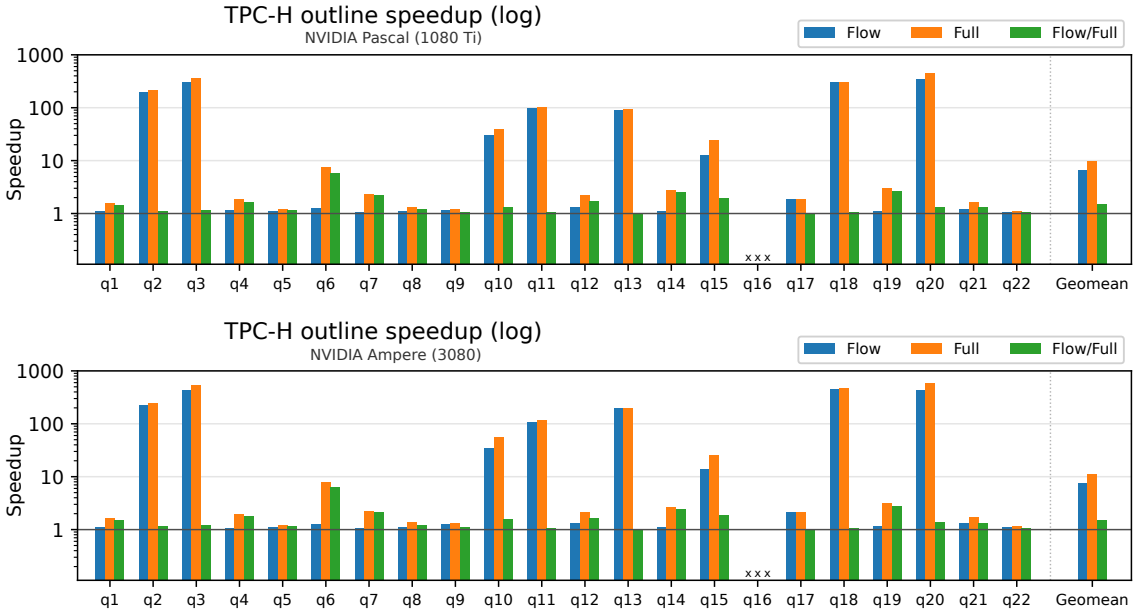


Figure 7.13 – Query speedup from outliner (flow: 1st pass; full: 1st + 2nd pass; flow/full: relative speedup from 2nd pass), compared to an unoptimized baseline with distinct kernels for each operation. Note the logarithmic scale.

The initial outlining phase targets kernels with consistent intermediate data size, removing the need for intermediate data materialization. In addition, we optimize list cell reductions by merging the subsequent `@raze` and producing a vector output. Initialization cost is thus significantly lower as the data is contiguous; a similar feel to the `@group` optimization. Together these effects significantly improve performance, especially the latter (e.g. q2, q3). Note that q16 does not support the naive or flow outlining as it requires computing a prefix sum on list cell data – unimplemented in our code generation. The full outliner produces kernels which avoid this step.

Outline optimization targets queries that reduce compressed output (q6, q19), as it avoids computation of the prefix sum. In addition, kernels which share input data are merged, more efficiently exploiting limited memory bandwidth. The speedup is less impressive than the first phase due to the oversize influence of data management, but still significant overall (~1.5 geomean). Despite being a greedy approach, we

observe no performance degradation on any supported query and the performance gains are also architecture independent. This does not hold in general, however, as the performance improvements from eliminating the memory accesses and prefix sum may be offset by decreased parallelism from compression. While it does not currently occur on our benchmarks, we observed this effect in earlier testing.

The impact of outlining is present both in execution and compilation, reducing the complexity and cost of the generated code. Shown in Figure 7.14, we note a significant decrease in compilation time from merging compatible kernels, further improving end-to-end performance. While the outliner does increase costs with each optimization level, the resulting kernels are notably fewer and much simpler to analyze and compile in subsequent steps. Further analyzed in Table 7.3, we present the number of outlined kernels for each query and outline mode (none, flow, and full), demonstrating the effectiveness of each phase at identifying compatible operations. Recall that q16 requires the full outliner to generate code.

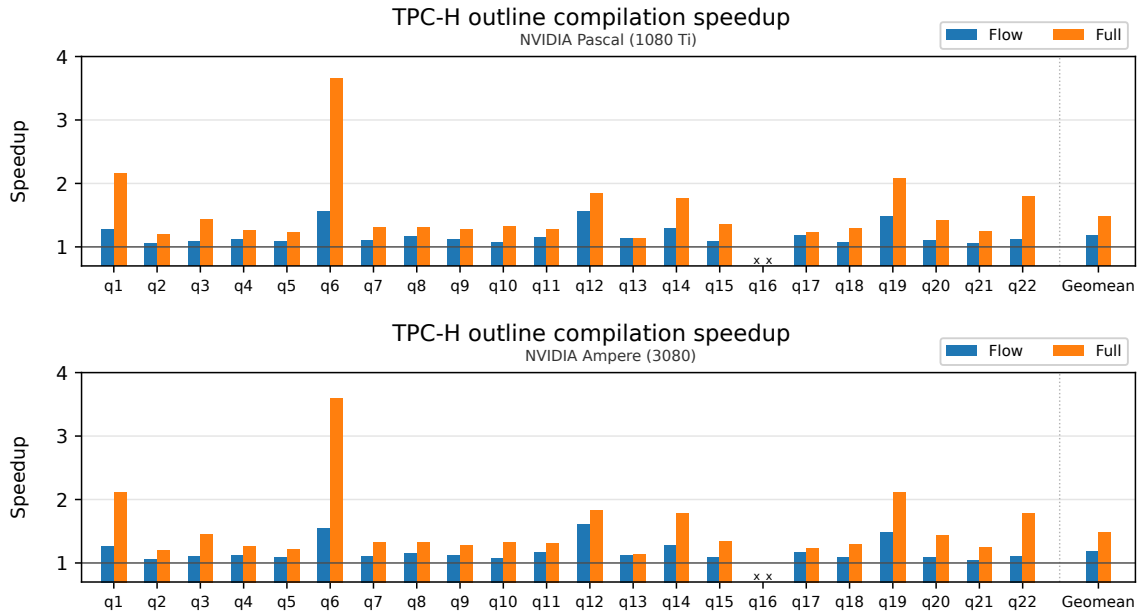


Figure 7.14 – Compilation speedup from outliner (flow: 1st pass; full: 1st + 2nd pass; flow/full: relative speedup from 2nd pass), compared to an unoptimized baseline with distinct kernels for each operation.

Table 7.3 – Number of kernels in each outliner phase, excluding libraries.

Q	None	Flow	Full
1	57	38	5
2	49	44	18
3	34	27	9
4	15	11	9
5	35	27	13
6	13	4	1
7	63	51	19
8	56	39	17
9	39	29	11
10	54	47	9
11	26	15	9

Q	None	Flow	Full
12	36	13	4
13	19	12	10
14	21	10	4
15	25	20	8
16	N/A	N/A	8
17	25	13	10
18	35	29	10
19	46	12	3
20	27	20	10
21	33	28	15
22	25	19	6

Performance improvements from outlining are query dependent, with the number of kernels not directly related. For example, despite only a minimal decrease in the number of kernels in phase 1 for queries q2 and q3 (see Table 7.3), its impact on performance is substantial (see Figure 7.13). As previously explored, this is due to the initialization required for list reductions, which can either operate on individual cells (unoptimized) or a contiguous buffer (optimized). The inverse is true for other queries (q6), where the second optimization pass has limited effect on the number of kernels but large impact on the execution. For these queries, sharing input data or removing compression is essential for performance. Further exploration of outlining must therefore consider factors beyond the number of kernels and consider data initialization, algorithmic impacts, and compilation time. In addition, for systems which consider GPU clusters and data scalability, we must account for data partitioning.

Register Allocation

Translating from intermediate representation with unlimited virtual registers to real machine code requires a register allocation scheme. We evaluate our approach described in Section 5.2.2, linear scan, by comparing the number of registers allocated against NVIDIA's assembler for the same kernels and at both optimization levels. Shown in Figure 7.15 are the relative number of registers used in the best and worst kernels, and geometric mean for the entire query. Values higher than 1 indicate we

use more registers, whereas values less than one show we use less.

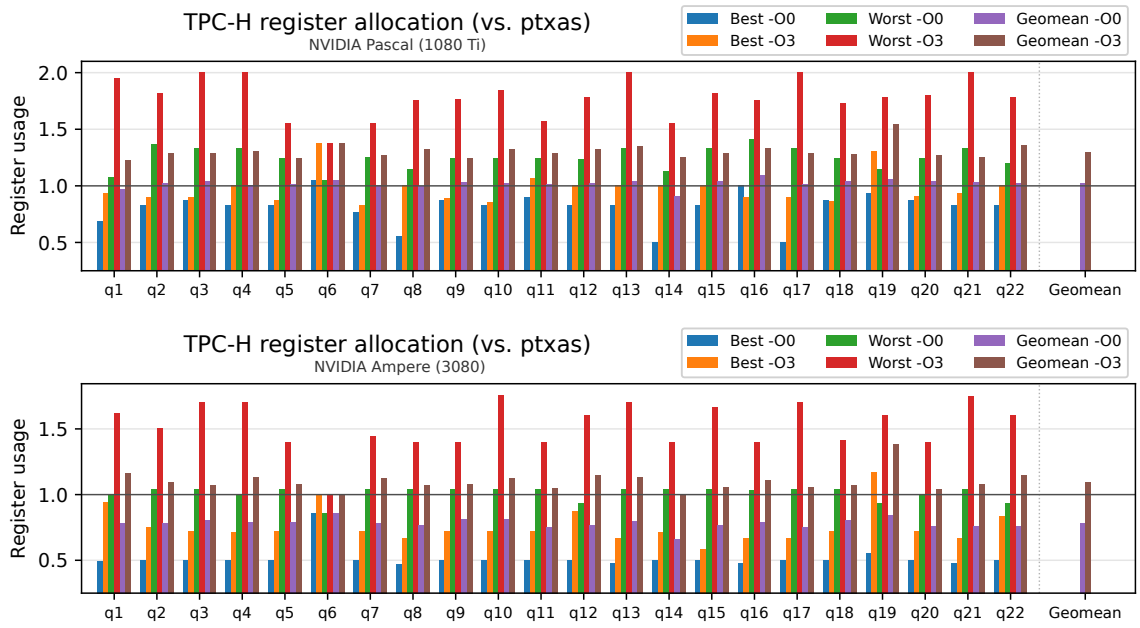


Figure 7.15 – Comparison of registers allocated by rNdN vs. ptxas. Values greater than 1 indicate more registers used by rNdN.

On average we allocate more registers than NVIDIA's optimizing backend, likely owing to the simplistic allocation strategy. We do, however, have comparable results to `-O0` on Pascal, and significant improvement on Ampere compared to their unoptimized assembler. In fact, our strategy is notably more competitive on the newer architecture with optimizations both enabled and disabled. Their allocation scheme for Ampere is thus not as efficient as Pascal, perhaps due to the philosophy of using "necessary" registers (see Section 5.2.2). Comparing the best cases, we see fewer registers in numerous kernels, while in the worst case our allocation is always within a factor of ~2. As register allocation algorithms are all heuristics, results naturally vary with the kernel and also depend on the code generation patterns and optimization strategies. The latter two may impact temporaries and live ranges. Interestingly, despite using more registers on average, our performance is only slightly lower than their optimizing backend, indicating that registers are unlikely to be the primary

bottleneck. This is despite the potential impact of register allocation on occupancy and parallelism. When considered with compilation time, we confirm that linear scan may not produce optimal allocations, but is an appropriate strategy for runtime use.

Scheduler

Instruction scheduling is our primary backend optimization, selecting an efficient instruction sequence and managing both fixed and variable-cycle dependencies. Shown in Figure 7.16, we measure the impact of list scheduling on kernel performance (excluding other execution time) and its implication for compile time and end-to-end execution. For comparison, we implemented a naive linear scheduler which does not reorder or pipeline instructions, waiting until completion before the next dispatch.

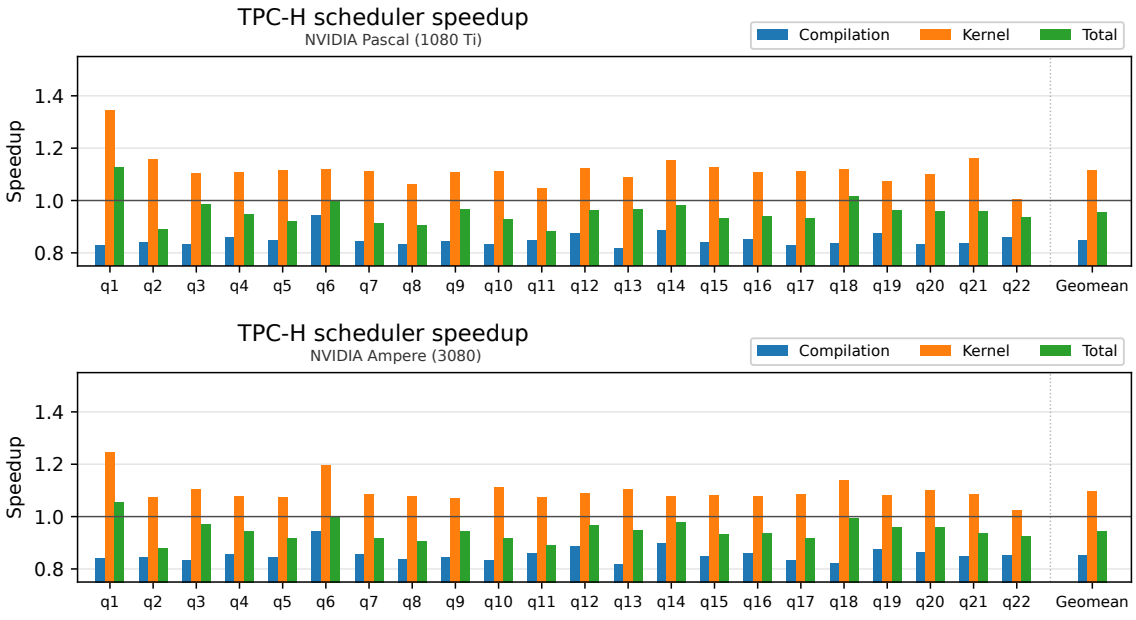


Figure 7.16 – Speedup of list scheduling compared to a naive implementation.

Compilation is unsurprisingly slowed using list scheduling, as it requires a basic block dependency analysis in addition to tracking availability of each instruction. Execution also follows the expected path, with kernels executing significantly faster over the naive approach. However, when considering end-to-end performance, we

found that the benefits to execution are outweighed by slowdowns in compilation in nearly all cases (e.g. q1). This holds for both architectures, which follow similar trends throughout, despite their differences in barrier implementation (recall that we do not exploit scoreboard registers in Ampere). Instruction scheduling may thus be appropriate for only cached scenarios, and requires further exploration to balance with compilation and end-to-end cost.

7.6 Summary

GPUs are an effective strategy for improving query performance, exploiting high degrees of parallelism to accelerate computation. They are, however, limited in end-to-end evaluation due to compilation and data overheads. We overcome these challenges through: (1) a balanced approach that trades minor slowdowns in execution for major speedups in compilation; and (2) optimized data management strategies.

Our evaluation shows significant improvements over both interpreted and compiled database systems on both CPUs and GPUs, with great performance on a variety of use cases. Importantly, we observe substantial speedup on end-to-end evaluations while still outperforming most cached comparison points. Further illustrating the balance of our approach, we see comparable cached results to NVIDIA's optimizing compiler with significantly less compilation overhead than the *non*-optimizing variant. Additionally, while data overheads are significant, they are sufficiently offset by speedups in computation and minimized by our allocation and transfer strategies. Lastly, we demonstrate how algorithmic optimizations for each operator and compiler techniques such as outlining improve query performance. Our detailed evaluation thus enables reasoned optimization in future systems, and supports the use of GPU databases in both cached and end-to-end scenarios.

Chapter 8
Related Work

Query compilation has long existed as a technique for improving the performance of database systems, while the use of accelerators like GPUs is a newer development. Our work combines and furthers these approaches, designing a system suitable for runtime compilation and execution of SQL queries on the GPU. It addresses overheads from data management and compilation by generating kernels with minimal materialization and maximum parallelism and translating to machine code with a purpose-built compilation pipeline – all while maintaining simplicity. We explore existing techniques for query compilation and GPU database systems in Section 8.1, and approaches for GPU compilation and assembly in Sections 8.2 and 8.3 respectively.

8.1 Databases

Database systems are the primary context and motivation for our work, with numerous optimizations proposed by the database community. rNdN is a compiler-driven approach that merges two key areas: (1) efficient query compilation discussed in Section 8.1.1; and (2) GPU database systems outlined in Section 8.1.2.

8.1.1 Query Compilation

Query compilation has long existed as an approach for improving performance in relational databases, avoiding the interpretation overhead. Ranging from high-level to machine code, the chosen target language trades between compilation times and implementation complexity. High-level languages are simple but slow, whereas low-level targets are complex but offer better performance. Although it seems clear that compilation can improve performance, its performance is not guaranteed in all cases (e.g. due to compilation overhead) [116].

Compiling queries to a high-level language is relatively simple, typically generating code for each operation in the query plan [121]. LegoBase takes this idea further by implementing the database itself in Scala *metaprogramming* to optimize the query in combination with its surrounding constructs [121]. This outperforms approaches which target queries directly or rely on general-purpose compilers, although overheads are measured in hundreds of milliseconds – a prohibitive cost for runtime compilation. Another high-level approach, Kernel IR from Red Fox, has been proposed as an intermediary between queries (LogiQL) and parameterized CUDA code templates [233]. While not explored in their work, compilation thus depends on the NVIDIA pipeline and would be a significant performance bottleneck. Voodoo has also been proposed as vector-based approach to target various architectures, generating OpenCL code [183]. As OpenCL is a high-level language similar to CUDA, we suspect compilation overhead is significant although not directly evaluated.

LLVM IR is a popular low-level target language [127], allowing faster compilation than traditional user-level programming languages while maintaining platform independence. Neumann extended the HyPer database system with LLVM code generation, generating simple operations directly and inserting calls to a pre-compiled C++ library for those that are more involved [152]. Compilation overhead is thus significantly reduced over high-level languages as it bypasses the frontend compiler and produces less code. Kohn et al. have also investigated the use of an interpreter for an LLVM IR inspired representation, executing each operator in a virtual machine and compiling and/or optimizing only when deemed "beneficial" [122]. An NVIDIA

variant of LLVM IR (NVVM IR [173]) has also been adopted by OmniSci in their GPU database [215]. Compiled to PTX at runtime and then assembled and loaded by the CUDA driver, compilation time is minimized compared to higher-level CUDA code, especially when combined with caching. Our initial design followed a similar idea, generating query programs in the PTX intermediate representation and bypassing heavy frontend compilers. In practice this proved insufficient, as it still depended on the proprietary assembler for binary generation. Our final approach thus employs an end-to-end, runtime-suitable pipeline with minimal compilation overhead.

Additional intermediate representations have also been proposed for database systems, extending existing languages ideas with database-specific constructs. HorseIR is one such approach, building on the principles of array-based languages to enable traditional compiler techniques on database queries [38, 37]. Also supporting MATLAB, this enables merging user-defined code into the query itself. Our approach is derived from HorseIR, extending the use of array-based languages to GPU databases. Although HorsePower does support GPU code through OpenACC directives (similar to OpenMP, see Section 8.2.4), its performance is limited by data transfers on the selected benchmarks – a concern not present in this work as we parallelize (nearly) the entire computation and have few intermediate transfers. Additionally, as the GPU is the primary execution unit, our approach optimizes kernels beyond simple directives and exploits additional control over parallelism.

On the lower-level, Flounder IR omits complex operations present in LLVM IR in favour of quality-of-life features to more easily support and optimize SQL queries with low overhead [69]. It has also been implemented in research database ReSQL [71]. Furthering the idea, Umbra IR is an low-level representation that captures the "intent" of database operations [117]. Implemented in the Umbra database system, it allows efficient translation from the execution plan and requires minimal optimization when lowering to machine code. Our approach is most similar to these systems, generating machine code without heavy compilers and using only necessary optimizations to reduce overhead. We extend this idea to GPU databases and PTX code, a previously unexplored field due to the closed-specification. Another recent runtime-optimized compilation approach uses *binary stencils* for efficiency [237].

8.1.2 GPU Databases

Since the introduction of general-purpose GPUs and the CUDA/OpenCL platforms, numerous GPU databases have been proposed. Prior to this, attempts at offloading database workloads required use of the programmable pipeline [82]. An early survey by Breß et al. (2014) provides an overview of the initial implementation techniques and identifies key areas for further research (e.g. data management, GPU-specific optimizations) [22]. More recently, several GPU database systems were found to have incomplete support for various SQL queries (e.g. some of TPC-H) and difficulty scaling to larger data sizes [45]. We begin by discussing existing database implementations, before exploring algorithmic optimization, heterogeneous processing, data management, operator pipelining, and miscellaneous extensions.

Seminal work by He et al. on GPUQP (also called GDB) implemented the first complete GPU database system in CUDA, with prior work focused on offloading specific operators [91, 92]. Each operator was built from efficient *primitives* that maximized parallelism and optimized memory accesses. Similar to our approach, strings are represented as indexes into a separate array. Existing database systems have also adopted GPU support, either porting each operator (e.g. PG-Strom for PostgreSQL [217]) or implementing a generalized parallelism strategy. In particular, Ocelot implements *hardware-oblivious* OpenCL operators for MonetDB, supporting both CPU and GPU parallelism [97]. Similarly, OmniDB (unrelated to OmniSci) uses kernel abstractions to decouple operator implementations from the hardware and reduce duplicate code [250]. Other work on GPU databases has explored porting the *virtual machine* to execute on the GPU itself [17], and the roles of data, hardware and operators on performance [245]. Our approach is unique in GPU database space in that we are compiler-first, database-second. We focus our attention on the design and implementation of an efficient compiler framework to exploit GPU parallelism, and rely on HorseIR for efficient expression of database queries. Compared to other work, we can therefore achieve significantly lower compilation overhead while other systems rely on comparatively heavy pipelines. rNdN is also designed to support general HorseIR programs beyond database queries and is not limited to SQL.

GPU databases have also progressed into commercial products, demonstrating their interest beyond research. As previously discussed, OmniSci exploits query compilation for performance on both CPU and GPU systems, with optimizations to reduce data transfers and support clusters [150]. On the other hand, BlazingSQL relies on an optimized library, RAPIDS.AI, for each component of the query plan [103]. They thus take two opposing strategies, similar to the divergence in CPU databases. Research has also explored the properties of a robust implementation [129], and the extension of an existing commercial database, DB2, to support GPUs [146].

Algorithms

Algorithm design is a notable area in GPU databases, with extensive research on the optimization of heavy operators. In particular we observe a focus on join, group/aggregation, sort and selection as they are most likely to benefit from acceleration.

Initial work on join translated algorithms originally designed for the CPU (e.g. sort-merge, hash), reusing optimized GPU *primitives* in each approach [92]. Subsequent research has explored the impact of more recent hardware [196, 198], extension to multi-GPU systems [179, 197], optimizations for data management [109, 206], and performance across hardware, algorithms and data [238]. Grouping and aggregation have also been explored, with hash aggregation preferable for most cases and group by sorting suitable only when the number of output groups is large and the data fits on the GPU [114]. Further exploration has concerned heterogeneous systems [220], and optimization across architectures [192]. Optimizations for selection have considered the impact of divergence [207], and numerous sorting algorithms have been designed or ported to the GPU [205, 81, 32]. There is thus significant potential for GPU-specific optimizations that remains unexplored in our work. In our system design we opted for a simplified hash join, group by sorting, and bitonic sort – all of which were selected with simplicity in mind. Optimizations focused on major sources of performance, including use of the memory hierarchy (e.g. shared memory), strength reduction (e.g. bitwise operations) and buffer management. Tuning for query and architecture properties like data sizes and cache was omitted for simplicity.

Heterogeneous Architectures

Although not considered in our approach, heterogeneous processing has been a recent area of interest, exploring whether a balanced co-processing approach yields better performance. While it seems obvious that using multiple processors can improve performance, other research has shown that data transfer costs can render it impractical [202]. Optimizing data transfers, effectively using each processor, and minimizing development overhead remain challenging problems [191]. Coupled architectures, where the GPU and CPU are "combined"[1], have also been of some interest due to the elimination of transfer cost. They have successfully optimized cache usage [95] and enabled a more granular decomposition and execution of each operator [94].

Query plans are traditionally optimized for CPU architectures, with only limited research on GPU systems [9]. Heterogeneous environments are more demanding, requiring balance between data transfers and relative operator performance. Multiple approaches have been proposed, each presenting a heuristic *cost model* that optimizes query plans to exploit multiple devices [25, 113]. One such query plan optimizer, HyPE [25], has been adopted in Ocelot [23] and CoGaDB [20]; the former being a GPU extension for MonetDB and the latter a purpose-built GPU implementation. Heuristics can either independently select the execution unit of each operator at runtime, or optimize across the entire plan [111]. A hybrid approach partitions query plans into subgraphs that allow reliable data size estimates, and optimizes each subgraph at runtime [112]. Other techniques for heterogeneous query planning include prioritizing data locations over operator performance and scheduling operators whose data dependencies are satisfied (i.e. topologically) [21].

Further research on HAPE [44] and HetExchange [43] has explored the use of architecture-specific operator implementations and a query plan extended with data management and device context switching. Code is JIT compiled for both CPU and GPU architectures using the LLVM framework and Proteus database system [115]. Other work on code generation in Hawk has proposed training an optimization framework for architecture-specific specialization of general-purpose implementations [24].

[1] https://en.wikipedia.org/wiki/AMD_Accelerated_Processing_Unit

Data Management

As a significant bottleneck, data transfers over the PCIe bus are important optimization targets. Data compression is one solution, reducing the size of data transferred [61, 194]. In heterogeneous systems, *approximate* results can also be computed on compressed data transferred to the GPU that is then *refined* on the CPU [182]. Alternatives approaches explore methods for keeping data in host memory until accessed by the GPU kernel using Unified Virtual Addressing (UVA [89]), and eliminating repeat transfers of identical data [189]. Newer hardware features like NVLink have also been shown to dramatically improve performance, although not available on our system [140]. Our approach transfers uncompressed data between the CPU and GPU as required, focusing instead on keeping data GPU-resident as much as possible. Exceptionally, string data uses dictionary encoding and is cached on the GPU device memory using a specialized kernel over UVA – a mixed approach.

String representation is also challenging on the GPU, as the traditional pointer approach requires a shared address space on both devices [17]. The Virginian database stores table data in one-or-more *tablets* where strings are represented as relative addresses into an associated pad [16]. This is extremely similar to our dictionary encoding approach (i.e. an index), although we use a global structure for all string data as it enables efficient cross-table equality. OmniSci likewise supports dictionary encoding [150]. Other work reorganizes characters between strings (called *pivoting*) to coalesce memory accesses, and evaluates porting and optimizing CPU algorithms for the GPU [208].

Pipelining

Pipelining operators is essential to best performance, eliminating the cost of intermediate data materialization [188]. This strategy is employed by the HyPer CPU database, accumulating operations between *pipeline breakers* that require materialization [152]. The analogous optimization on the GPU, kernel fusion, combines individual kernels into a larger pipeline and reduces expensive device memory accesses. GPU databases either match pre-defined patterns on the query plan (e.g.

HippogriffDB [135, 68], PG-Strom [217]), employ pipeline breakers from HyPer (e.g. HetExchange [43]), or use other compiler driven techniques like Kernel Weaver (e.g. OmniSci [215], Red Fox mentions the possibility [233]). Kernel Weaver was introduced as a heuristic-aided approach to effectively fuse data-dependent kernels in a database system provided that they have identical thread layout and inter-thread dependencies can be satisfied [232]. Wu et al. also considered its interaction with *kernel fission* [234], an approach similar to tiling under which the working set is split and data may be transferred at the same time as computation [88].

Also in database systems, Funke et al. proposed *fusion operators* to pipeline compatible query steps, each of which is compiled to a separate *compound kernel* [68]. Basic queries are handled automatically, but more complex cases require user direction. Further approaches to general kernel fusion have included combinatorial approaches [227], graph algorithms like minimum cut [187], and its application to nested queries [63]. An alternative strategy in GPL uses more recent OpenCL features (note that this work refers to *channels* though they are typically called *pipes* in OpenCL[2]) to efficiently transfer data between concurrent kernels [178].

Our implementation is built on HorseIR, benefiting from an array-based representation that allows us to easily identify data dependencies, synchronization, and parallelism. An existing shape analysis-based technique for loop-based fusion in HorseIR was proposed for CPU implementations [39], and we extend its application to GPUs. In particular, we consider the impact of limited synchronization, the parallel programming paradigm, and the mapping to threads and blocks. Compared to other kernel fusion techniques, we exploit an abstract representation (i.e. geometries) rather than thread layouts, and generalize beyond database applications and query plans. We also consider *compatible* kernels rather than exact matches, and support general fusion without the use of compiler hints. Our approach is also greedy, compared to the heuristic-driven approach of Kernel Weaver.

While pipelining may improve performance by reducing intermediate data materialization, fused kernels are more frequently impacted by thread divergence due to

[2] https://www.youtube.com/watch?v=_0RtAKeRl00

their increased branching complexity [70, 177]. Pyper [177] and DogQC [70] mitigate this effect through careful (possibly dynamic) mapping of data to each thread. Pyper also explores trading fusion for occupancy, while we note DogQC targets performance by efficiently pipelining *simple operators* [70] – the same as in our approach.

Miscellaneous

Our work focuses on the compilation and execution of individual analytical queries, but GPU databases have been extended to other contexts. In particular, we note exploration of concurrent queries [229, 249, 189], transactions (OLTP) [93], nested query evaluation [63], query plan optimization (i.e. accelerating the optimization itself) [96], and SSD storage [249]. Additionally, FPGAs have been explored as a possible execution platform [60].

8.2 Compilers

Due to the inherent complexity of GPU programming, automatically generating efficient parallel code from a higher-level representation has been the focus of much research. We discuss approaches to automatically detecting parallel algorithms in Section 8.2.1, purpose-built languages and intermediate representations for parallel programs in Section 8.2.2, language extensions and frameworks in Section 8.2.3, and directives-based approaches in Section 8.2.4. We lastly discuss some existing approaches to lower-level GPU compilers in Section 8.2.5.

8.2.1 Automatic Parallelism

Automatic parallelism is widely accepted as a challenging problem, although it is possible in constrained environments. In the GPU space, approaches typically identify known sources of parallelism using patterns or rules [134, 77, 209], are restricted to well-defined subsets of a programming language [65], or use polyhedral [119] or traditional [105] loop parallelization strategies. The latter is adopted in the ALPyNA Python framework which detects loops free of data-dependencies both statically *and*

dynamically [105]. Compiler analyses can also be used to annotate source programs that are subsequently compiled using templates (referred to as *skeletons* in some work) [156, 157]. There has also been exploration of speculative approaches, where *possibly-data-parallel* kernels are offloaded and their execution verified [199].

Our approach to automatic parallelism is most similar to either pattern-based work (a pattern of size 1), or those which target restricted subsets of a language. As HorseIR provides a well-defined set of built-in functions, we can decide on each parallelism strategy ahead-of-time and implement the necessary code generation templates in our compiler. This is ideal for runtime compilation as it requires no expensive analyses (e.g. data-dependence) to determine parallel viability.

8.2.2 Languages and Intermediate Representations

Programming languages and intermediate representations have also been designed to support parallel computation. LIFT is one such approach, generating OpenCL code by translating from a high-level to a low-level functional representation using *rewrite rules* [212, 213]. Automated rewrite tuning and *semantic information* in the IR yield portable and efficient code. Array-based intermediate forms like VRIR have also been explored, supporting runtime shape analyses and outlining of *potentially interesting regions* of loop and vector operations [73]. It is also used as the intermediate representation for Velociraptor, an *embedded* compiler that generates parallel code for *outlined regions* at runtime, possibly targeting the GPU [74]. Intermediate representations have also been designed for distributed environments [27], and as building blocks to parallelize domain-specific languages [28]. We elected to use an array-based intermediate representation due to its implicit parallelism and similarity to vector processing, inspired by the use of outlined regions and the accompanying shape analysis in Velociraptor. Our work extends both ideas by proposing an ahead-of-time approach to define efficient kernels in arbitrary array-based programs through symbolic shape analysis.

8.2.3 Language Extensions and Frameworks

Existing languages have also been augmented with GPU-suitable parallelism constructs. An extension to Java, the Lime language [15] is a general approach to support multiple architectures, enabling easy generation of optimized OpenCL for GPUs [58]. Liquid Metal takes the idea further and compiles each operation for multiple architectures, selecting the exact device at runtime [14]. C and C++ have also been extended to support common parallel programming patterns either through low-level primitives [137] or skeletons [200]. The latter may fuse skeletons within a basic block and duplicate computation to avoid intermediate results.

Numerous projects have also extended GPU support to Python. Parakeet provides a library of parallel functions that operate on NumPy arrays, using pattern-based fusion and PTX code generation for efficient compilation and execution [195]. On the other hand, PyCUDA and PyOpenCL embed kernel definitions in the code and may benefit from runtime information [120], while Copperhead is a Python subset augmented with parallel *primitives* and optimized using fusion [31]. Similar to our approach, operations may be fused according to data dependencies and *completion requirements* (analogous to synchronization). Other higher-level languages like Ruby have also been extended, using skeletons for supported operations and fusion for element-wise computation [142, 210, 211]. Alternately, a Haskell-based approach employed an array-based representation to expose GPU parallelism [35]. Notably, their runtime reduces compilation and data transfer overheads through *asynchronous* transfers. Compilers for StreamIt (stream programming) [99] and LINQ (.NET) [193] have also been proposed, both of which support fusion.

Parallel programming frameworks have also been proposed, implementing a variety of GPU algorithms. Thrust [174] and CUB [172] from NVIDIA are two such solutions, with the former possessing a C++ interface and the latter exploiting CUDA features. Alternatively, CuPy is "drop-in replacement" for NumPy that uses libraries and user-defined kernels to offload computation [175]. DelayRepay also targets NumPy, using *delayed execution* to build an AST with each function call and compiling/executing only when required [149]. Fusion may thus be applied during code generation.

8.2.4 Directives

A complement to automatic translation, user-provided *directives* indicate parallel regions to the compiler and remove the need for complex analyses. As a well known implementation for CPUs, OpenMP has been adapted in multiple projects [131, 130, 56, 155, 230, 100], each defining an equivalent GPU parallel pattern. A GPU-supporting standard, OpenACC, has also been developed and implemented for the specific needs of accelerators (e.g. data transfers) [176, 132]. hiCUDA takes a similar approach, exposing GPU programming details like kernels and data transfers for CUDA [86]. Directives have also been included in the PGI compiler [231], Python-based solutions [72], stream programming [144], and can also be introduced automatically by compiler analyses for certain kinds of parallelism [145, 11, 157].

8.2.5 Lower-Level Compilers

Lower-level compilers have also been explored, particularly those handling intermediate representations. gpucc from Google is an open-source alternative to the CUDA frontend compiler, generating optimized PTX code [236]. They describe an optimization pipeline that increases performance over the NVIDIA compiler with lower compilation overhead. Also open source, the Ocelot project (distinct from Ocelot database) is a low-level compiler that translates PTX code for execution on multicore CPUs [51]. It has been extended in Caracal [54] to support generation of AMD's low-level representation, the Compute Abstraction Layer [5]. A similar project to Ocelot, Twin Peaks, has explored OpenCL translation for CPUs [85]. Our approach to low-level compilation is most similar to gpucc, in that we target PTX in our frontend compiler. However, we start from a more constrained representation (HorseIR), and thus do not need to perform significant optimization to achieve reasonably efficient code. In fact, our PTX code is generated from templates which are already optimized for our context.

8.2.6 Optimization

GPU optimization is a significant topic in compiler research, with techniques typically addressing memory accesses and parallel mappings [242]. We also note research on runtime-optimized kernels [148], and the combination of static annotation with dynamic optimization for PTX code [128]. Although optimization is limited in our approach, opting instead for efficient templates, it remains interesting future work.

8.3 Assembler

Backend GPU compilers or assemblers have typically been closed-source implementations developed by the hardware manufacturers themselves. This trend has recently been reversed, with AMD [7, 8], Intel [36] and Apple [141] all releasing varying degrees of information on their LLVM extensions. Our solution targets NVIDIA hardware, which is comparatively opaque besides reverse-engineering efforts. We present existing work on GPU assemblers in the following sections and decompose the open-source manufacturer pipelines into their components. In particular, we focus on binary generation for NVIDIA architectures in Section 8.3.1, register allocation in Section 8.3.2, instruction scheduling in Section 8.3.3, and control-flow structuring in Section 8.3.4.

8.3.1 NVIDIA Architecture

As a tightly guarded secret, NVIDIA does not currently discuss the instruction set beyond mnemonics and disassembly tools [162]. Much of the existing research has therefore focused on reverse-engineering, laying the groundwork required for our project. In comparison, AMD has release significant architecture details and the complete instruction sets on their GPUOpen website [8].

Multiple open-source assemblers targeting NVIDIA architectures have been developed in recent years, each supporting a collection of compatible GPU families. Binary and instruction formats are reverse-engineered, providing a convenient method for modifying existing CUDA binaries or writing low-level programs directly in SASS. In

order of architecture recency[3], implementations include: Decuda for G80 [224], asfermi for Fermi [246], KeplerAs for Kepler [251], MaxAs for Pascal and Maxwell [83], and TuringAs for Turing, Volta and Ampere [239]. While manual reverse-engineering is viable for individual architectures, more extensible approaches include *differential analysis* (e.g. KeplerAs [251], DecodingCUDA [90]) and constraint solving using matrices of instruction variations (e.g. CuAssembler [46]). CuAssembler in particular presents a comprehensive implementation of binary generation across platforms, although instruction support depends on the completeness of the solver input [46]. Despite their success, assemblers are limited to the SASS realm, perform little optimization besides MaxAs [83] (discussed in later sections), and may be designed for specific contexts or individual kernels (e.g. TuringAs for convolutions [239]).

Our solution builds on these prior works, extending their principles to query compilation and execution. Based on MaxAs [83], we develop, formalize and evaluate a complete PTX to SASS translation pipeline that allocates registers, performs instruction scheduling, structures control-flow, and generates a relocatable binary. Additionally, we support multiple distinct architectures and instruction sets and keep compilation overhead low enough for use in runtime systems – a unique goal. Concurrently developed, Yan et al. have also proposed a preliminary, open-source, LLVM-based implementation for compiling LLVM IR to SASS (supporting Volta, Turing, and Ampere) without the use of NVIDIA's proprietary pipeline [240]. Similar to our approach, they implement instruction scheduling and control-flow flattening to improve performance, although we focus on runtime efficiency for just-in-time use and sidestep ahead-of-time frameworks due to their cost.

Other research on GPUs reveals low-level architecture details on memory and caches [107, 106], branching implementations [124, 143], instruction latencies [107, 106, 13], or low-level tools for interacting with NVIDIA hardware (including assembly/disassembly and instruction formats) [123]. Simulators for both AMD and NVIDIA architectures have also been developed [118, 223], and a translation layer

[3] https://nouveau.freedesktop.org/CodeNames.html

from Maxwell SASS to other machine code is implemented in a Nintendo Switch emulator [247][4] (the Nintendo Switch uses the Maxwell architecture [216]). Of particular note is the Nouveau driver in the Mesa project, which implements a complete compilation pipeline (including register allocation, scheduling, and optimization) from its own intermediate representations to SASS [64]. This is equivalent to our work, although we target compute applications through PTX rather than graphics.

8.3.2 Register Allocation

GPU-specific register allocation algorithms are typically designed with both vector and scalar registers in mind, as serialized execution from divergent branches introduces additional scalar dependencies not caught by liveness analysis [110, 42]. Consequently, Kalra proposed a graph colouring approach for AMD GPUs that separately allocates scalar values using additional control edges [110]. Similarly, the Intel production compiler uses an *augmentation analysis* to add additional constraints to the interference graph [42]. Their approach also minimizes bank conflicts and false dependencies, with variables contained within a basic block allocated using linear scan and the remaining global variables relegated to graph colouring. As our implementation allocates vector registers, we can safely ignore these dependencies. For embedded GPUs with limited register resources, efficient allocation is essential. You and Chen introduce an extension to linear scan, *element-based register allocation*, that independently considers the live range of each component (i.e. x, y) for vector allocations [243]. This approach allows more efficient register usage as some vector components may be reused before others.

MaxAs, an assembler for NVIDIA Pascal and Maxwell GPUs, implements a basic register allocator for code without control-flow [83]. It focuses on efficient use of register banks, reducing conflicts through its assignment policy and use of register caching. Our register allocator is based on linear scan [185], computing live intervals for the complete control-flow and adapting the allocation to support register-pairs

[4] https://www.reddit.com/r/hardware/comments/bsmjpe/ptx_instruction_latencies_across_nvidia/

and alignment (similar to what is done in Intel's graph colouring approach [42]). We therefore support general GPU code, but ignore low-level allocation details like bank conflicts due to the compilation overhead. Other production compilers like Apple's have discussed the importance of balancing register pressure and occupancy in their approach, while minimizing costly spills [141]. Although no absolute timings are presented, they indicate that instruction selection is heavier than register allocation – the opposite of our pipeline. Note that other approximations for live variables have been proposed for CPU databases, achieving *near* linear complexity [122, 153]. The Nouveau allocation scheme is based mainly on interference graphs and is unlikely to be suitable for runtime use [64].

8.3.3 Instruction Scheduling

Efficient instruction scheduling requires collaboration between both software and hardware, balancing resource usage with increased parallelism.

Software

Software instruction scheduling produces an efficient ordering, possibly encoding metadata for hardware use (NVIDIA architecture). MaxAs implements list scheduling [75] for Pascal and Maxwell architectures, optimizing user-defined schedulable sections. Their heuristic is based on (in order): fixed stall counts, dual issue capability, *mixing* functional units, and the number of dependencies [83]. Our approach builds on and formalizes this work, systematically supporting multiple architectures and general control-flow. Additionally, we automatically insert variable-cycle dependency barriers (possibly using scoreboard registers) and use a tuned heuristic that prioritizes the *expected* stall. The latter is important for our context given the prominence of high-latency memory accesses in database queries. Additionally, while MaxAs supports dual issue and register reuse flags, we omit these optimizations to improve compile time. Note that MaxAs allocates registers after scheduling, whereas we implement the opposite – an arbitrary choice given their cyclic dependency.

The Nouveau driver scheduler is likewise derived from MaxAs[5], although it does not consider instruction reordering or throughputs [64]. Scheduling is on a per-block basis, tracking dependencies and computing stall counts by recording the available time for each register rather than using an explicit dependency graph. Barriers are inserted automatically (no scoreboarding), and they consider interactions between basic blocks. TuringAs has explored manual optimization using the yield flag and high-latency instruction placement [239]. Our use of the yield flag corresponds to their "natural yield strategy", although we chose this approach primarily for simplicity.

Gong et al. propose TwinKernels, a compile-time scheduling approach that produces two distinct instruction schedules for each kernel [78]. Warps are assigned to either of the two implementations during execution, reducing contention from high-latency operations. Compile-time *trace scheduling* has also been investigated, using *speculation* to move (high-latency) instructions to earlier basic blocks and minimize divergent execution [104]. Synchronization instructions preclude certain reorderings, similar to our idea of schedulable sections. Additionally, the authors note that predication may produce longer traces, an approach used in our work to expose additional reordering possibilities, though they rely on NVIDIA's pipeline for this optimization.

Other instruction schedulers merge the problem with register allocation, balancing register pressure and increased ILP. Goodman and Hsu proposed both an adaptive solution that uses multiple scheduling heuristics and a DAG-based register allocator that reduces false dependencies [80]. Shobaki et al. employ a *branch-and-bound* algorithm, prioritizing occupancy and register usage before selecting the instruction order [204]; a technique augmented with graph transformations to reduce the search space [203]. Occupancy is also used in the LLVM list scheduler for AMD GPUs, amongst other GPU properties [7, 204]. Similarly, Intel selects their scheduling heuristic based on register pressure and ILP *thresholds* [36], and Apple uses "standard scheduling" that balances register pressure and ILP [141]. Other approaches include modified interference graphs [181], or the combined interaction with loop unrolling [53].

[5] https://gitlab.freedesktop.org/mesa/mesa/-/commit/f519c47f7d47d88ecf3b5e8f28fdffaa12f684d3

Hardware Support

Hardware approaches to warp scheduling are also important for performance, reducing stalls and avoiding divergence. Yu et al. propose *Stall-Aware Warp Scheduling*, rescheduling frequently stalled warps for later execution and allowing others to execute without contention [244]. Gong et al. have also explored using compiler hints to *hide* high-latency operations using *out-of-order execution* [79]. This improves performance over compile-time only approaches as the hardware can schedule based on *actual* stall counts rather than *static* estimates. Dynamic assignment of threads to warps may also improve performance by reducing divergent execution [67, 66]. Alternatives to *round-robin warp scheduling* that increase parallelism have also been explored [41]. As a software-only system, we rely on existing hardware properties for performance.

8.3.4 Control-Flow Structuring

Unstructured control-flow is challenging for compiler writers, from performance to hardware support. Structuring control-flow can simplify code generation and expose optimization opportunities, but is a complex process in itself. Zhang and D'Hollander introduced *hammock graphs*, "single-entry, single-exit regions" for structuring control-flow using a set of transformations [248]. This work has been extended to GPU contexts, some of which require structured control-flow (e.g. AMD IL [6]) [55]. Similarly, Wu et al. have proposed a transformation-based approach to structuring, analyzing its performance on *real* programs and reinforcing the importance of the reconvergence point [235]. Reissmann et al. demonstrated that re-execution of code duplicated during structuring can be avoided by producing *tail-controlled loops* and *well-nested control-flow* [190]. Linearization has also been proposed, reducing code expansion by structuring control-flow as "if-then statements" of predicated basic blocks [10].

While powerful, structured control-flow is often inefficient for unstructured programs. Diamos et al. introduce *thread frontiers* to manage the execution of divergent threads, eagerly reconverging when divergent paths coincide at the same basic block and avoiding repeated execution [50]. Less stringent approaches to structuring like reconvergence CFGs also reduce code duplication, allowing for unstructured branching

where "one of the successors is a post-dominator" (i.e. the reconvergence point) [228]. Other approaches like *melding* reduce thread divergence by restructuring branches that share common elements [201].

Our approach to control-flow structuring is much more limited, requiring a well-structured graph from the code generation stage. We thus proposed a greedy algorithm for recovering typical loops (including break), and if-else control-flows without any code duplication and reconverging at the immediate post-dominator. This differs from Apple's approach which supports unstructured control-flow through basic structuring and typical basic block duplication [141]. They also support flattening of branch structures to expose additional scheduling opportunities, similar to branch inlining used in our work. The Nouveau driver also uses flattening, replacing explicit branch instructions with predication [64].

Chapter 9
Conclusion and Future Work

Query compilation for GPUs has traditionally been limited by hardware capabilities, the design of parallel friendly algorithms, and the high cost of compilation. We propose a new GPU database system, rNdN, that challenges this status quo through a compiler-first design. Our approach extracts parallelism from an array-based query representation, outlines efficient kernels that minimize data materialization, and translates through multiple levels of intermediate representation to produce an assembled binary. By adopting simplistic yet efficient algorithms for each database operator and performing only necessary optimization, we significantly increase performance over the existing approaches in end-to-end evaluation without overly sacrificing the computation itself. This balanced approach is a unique characteristic in the GPU database space, with prior work depending on expensive manufacturer developed compilers. We can thus outperform comparison systems on both the CPU and GPU, interpreted and compiled, in contexts that frequently change data or queries, or are short running. This addresses two key limitations of previous approaches, data transfers and compilation, evaluated on a variety of queries. Further research on GPU databases can thus extend our runtime-optimized compiler and execution engine to other contexts, with limitations and future work discussed in more detail in Sections 9.1 and 9.2 respectively.

9.1 Limitations

As a GPU database rNdN has great performance potential, but its generality to other forms of computation and applicability to other platforms remains unknown. We identify 4 key limitations of our approach:

Algorithmic: As a guiding principle of our implementation, we opted for general-purpose and intuitive algorithms over intensive optimization. On the selected benchmark suite this was effective, but the success of each database operator implementation is well known to be data-dependent (e.g. hash vs. sort grouping). Widespread adoption is thus limited by our selected implementations.

Database oriented: Our approach is database-specific, targeting short-running queries and improving performance through reduced compilation overhead. It remains to be seen if the selected optimizations are sufficient for more general contexts or if further components are required. In particular, we suspect that our approach is more well-suited for data-intensive rather than computation heavy applications (e.g. scientific computation).

Hardware specific: Computation is accelerated on consumer-grade GPUs, relying on their widespread access for impact. As the adoption of other specialized devices (e.g. TPUs[1], FPGAs) increases, or as GPUs evolve or are replaced, we are unsure if the compilation techniques we selected will be as successful.

Data scalability: As a simplifying assumption, we supported data sizes that fit within the confines of GPU device memory. As data continues to scale and memory sizes increase, the impact of runtime compilation techniques is unclear. In particular, the overhead of ahead-of-time systems may become negligible and negate the usefulness of our approach, or further optimizations may be required to handle memory constraints.

[1] https://cloud.google.com/tpu/docs/tpus

9.2 Future Work

rNdN follows a pragmatic design, focusing on reasonably effective techniques at every step rather than intense optimizations. Key future work could therefore investigate further optimization of query compilation and execution, support for more general scientific computation, and extension to scaling data size and clusters.

Query optimization: Our approach uses CPU query plans from the HyPer database [152], translated to HorseIR by the HorsePower project [37]. Their effectiveness for different architectures is unclear, and from preliminary evaluation we suspect GPU-specific plans may better exploit the parallel paradigm and hardware characteristics.

Library functions: Library functions are currently considered as opaque boxes, limiting their interaction with other operations in the query. Further exploration could therefore investigate complete or partial pipelining in the outliner design, removing unnecessary materialization. As data sizes also impact the performance and resource usage of outlined kernels, heuristics for compatibility may also be beneficial (e.g. Kernel Weaver [232]).

Algorithmic optimization: The performance of certain operators depends on query characteristics and data sizes, requiring further investigation to select the best option. In particular, group by sorting is known to be inefficient for some queries, and may be replaced by other algorithms like hash aggregation [114].

GPU optimization: Optimization was a secondary concern in our design, instead implementing intuitive approaches that applied across architectures. Finer-grained GPU characteristics are thus largely ignored and require further consideration. In particular, we note the possibility of bank-aware register allocation [42, 83] and scheduling [83], caches and shared memory, and the use of scalar (i.e. uniform) operations and registers. In addition, the integration of occupancy into our heuristics may yield better performance, which may require reordering or combining the scheduling and register allocation phases.

JIT compilation: Currently, our system pre-compiles all kernels before execution, limiting the amount of runtime information. This was initially due to the high fixed-cost of NVIDIA's assembler for each invocation; a cost that we eliminated in our design. We therefore propose adapting our pipeline into a true JIT compiler that uses runtime information to produce (or re-compile) higher performing code. Additionally, it is now possible to hide our compilation time with data transfers, an approach taken in a Haskell extension for GPU programming [35].

General-purpose computation: Scientific computation is of significant interest, particularly with the support of MATLAB translation to HorseIR [37]. We could therefore extend our approach to more general programs, and consider the suitability of our limited optimization pipeline to other domains.

General-purpose PTX: Our assembler is currently limited to the code generation patterns of the frontend compiler, and by extension HorseIR. Extending its use to a more complete set of PTX programs is intriguing, as we can evaluate the impact of our approach on GPU programs beyond array-language built-in functions. Further implementation and evaluation of code generation templates, register allocation (possibly requiring spilling), and scheduling would be required, as is support for unstructured control-flow. rNdN would therefore be a complete drop-in replacement for runtime contexts beyond database queries.

Data scalability: Our approach addresses compilation time for short queries, specifically those whose data fits on device memory. Its impact is therefore relatively significant, as the data transfer and computation costs are reasonably low. However, as data sizes increase and execution becomes correspondingly longer, our approach has less relevance – though we still produce high performance code that is comparable to the proprietary alternative. Additionally, we must contend with different growth rates of data vs. device memory, which may require partitioning strategies to compute the full output. Future work could therefore evaluate the effectiveness of rNdN on varying data and device memory sizes (e.g. TPC-H scale factors), or apply further tuning to achieve on-par performance

to NVIDIA's optimizing compiler. The latter is particularly interesting, as we could improve performance of both short and long queries. Preliminary experiments show the increasing importance of data transfers as the input scales.

Multi-GPU support: GPU clusters are prevalent in high-performance computing, though we targeted consumer devices in our approach. It is therefore interesting to consider multi-GPU systems, partitioning the data and computation. Adapting the outliner to support multiple targets is one conceivable approach, using heuristics to select the best placement.

Bibliography

[1] Bison - GNU Project - Free Software Foundation, December 2021. URL: https://gnu.org/software/bison/.

[2] GitHub - westes/flex, December 2021. URL: https://github.com/westes/flex.

[3] GitHub - tensorflow/tensorflow, February 2022. URL: https://github.com/tensorflow/tensorflow/.

[4] GitHub - terralang/terra, February 2022. URL: https://github.com/terralang/terra/.

[5] Advanced Micro Devices, Inc. Compute Abstraction Layer (CAL), Dec 2010. URL: https://developer.amd.com/wordpress/media/2012/10/AMD_CAL_Programming_Guide_v2.0.pdf.

[6] Advanced Micro Devices, Inc. AMD Intermediate Language (IL), Oct 2011. URL: http://developer.amd.com/wordpress/media/2012/10/AMD_Intermediate_Language_(IL)_Specification_v2.pdf.

[7] Advanced Micro Devices, Inc. GCN native ISA LLVM code generator — ROCm documentation 1.0.0 documentation, Aug 2021. URL: https://rocmdocs.amd.com/en/latest/ROCm_Compiler_SDK/ROCm-Native-ISA.html.

[8] Advanced Micro Devices, Inc. Let's build everything - GPUOpen, Feb 2022. URL: https://gpuopen.com/.

[9] Adnan Agbaria, David Minor, Natan Peterfreund, Eyal Rozenberg, and Ofer Rosenberg. Overtaking CPU DBMSes with a GPU in whole-query analytic processing with parallelism-friendly execution plan optimization. In *ADMS/IMDM@VLDB*, 09 2016. doi:10.1007/978-3-319-56111-0_4.

[10] Jayvant Anantpur and Govindarajan R. Taming control divergence in GPUs through control flow linearization. In Albert Cohen, editor, *Compiler Construction*, pages 133–153, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[11] José Andión, Manuel Arenaz, François Bodin, Gabriel Rodríguez, and Juan Touriño. Locality-aware automatic parallelization for GPGPU with OpenHMPP directives. *International Journal of Parallel Programming*, 44(3):620–643, Jun 2016. doi:10.1007/s10766-015-0362-9.

[12] Austin Appleby. GitHub - aappleby/smhasher, Jan 2016. URL: https://github.com/aappleby/smhasher.

[13] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan J. Eidenbenz. Instructions' latencies characterization for NVIDIA GPGPUs. *CoRR*, abs/1905.08778, 2019. URL: http://arxiv.org/abs/1905.08778, arXiv:1905.08778.

[14] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 271–276, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2228360.2228411.

[15] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, page

89–108, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1869459.1869469.

[16] Peter Bakkum and Srimat T. Chakradhar. Efficient data management for GPU databases. Technical report, NEC Laboratories America, 2012.

[17] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 94–103, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1735688.1735706.

[18] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery. doi:10.1145/1468075.1468121.

[19] Piotr Bialas and Adam Strzelecki. Benchmarking the cost of thread divergence in CUDA. In Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, Konrad Karczewski, Jacek Kitowski, and Kazimierz Wiatr, editors, *Parallel Processing and Applied Mathematics*, pages 570–579. Springer International Publishing, 2016.

[20] Sebastian Breß. The design and implementation of CoGaDB: a column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14:199–209, 11 2014. doi:10.1007/s13222-014-0164-z.

[21] Sebastian Breß, Henning Funke, and Jens Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1891–1906, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2882936.

[22] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. *GPU-Accelerated Database Systems: Survey and Open Challenges*, volume 8920, pages 1–35. Springer Berlin Heidelberg, 12 2014. doi:10.1007/978-3-662-45761-0_1.

[23] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *Proc. VLDB Endow.*, 7(13):1609–1612, August 2014. doi:10.14778/2733004.2733042.

[24] Sebastian Breß, Bastian Köcher, Henning Funke, Tilmann Rabl, and Volker Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6), 09 2017. doi:10.1007/s00778-018-0512-y.

[25] Sebastian Breß and Gunter Saake. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMS. *Proc. VLDB Endow.*, 6(12):1398–1403, aug 2013. doi:10.14778/2536274.2536325.

[26] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994. doi:10.1145/177492.177575.

[27] Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 194–205, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854042.

[28] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *2011 International Conference*

on Parallel Architectures and Compilation Techniques, pages 89–100, 2011. doi:10.1109/PACT.2011.15.

[29] John Burgess. RTX on – the NVIDIA Turing GPU, Aug 2019. URL: https://old.hotchips.org/hc31/HC31_2.12_NVIDIA_final.pdf.

[30] John Burgess. RTX on — the NVIDIA Turing GPU. *IEEE Micro*, 40(2):36–44, 2020. doi:10.1109/MM.2020.2971677.

[31] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, page 47–56, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1941553.1941562.

[32] Daniel Cederman and Philippas Tsigas. GPU-Quicksort: A practical quicksort algorithm for graphics processors. *ACM J. Exp. Algorithmics*, 14, jan 2010. doi:10.1145/1498698.1564500.

[33] Stefano Ceri and Georg Gottlob. Translating SQL into relational algebra: Optimization, semantics, and equivalence of SQL queries. *IEEE Transactions on Software Engineering*, 11(4):324–345, 1985. doi:10.1109/TSE.1985.232223.

[34] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, page 98–105, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/800230.806984.

[35] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, page 3–14, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1926354.1926358.

[36] Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, Thomas Raoux, and Konrad Trifunovic. IGC: The open source Intel graphics compiler. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 254–265. IEEE Press, 2019. doi:10.1109/CGO.2019.8661189.

[37] Hanfeng Chen. *HorsePower: An Array-based Optimization Framework for Query Processing and Data Analytics*. PhD thesis, McGill University, Feb 2021. URL: https://escholarship.mcgill.ca/concern/theses/2j62s947s.

[38] Hanfeng Chen, Joseph Vinish D'silva, Hongji Chen, Bettina Kemme, and Laurie Hendren. HorseIR: Bringing array programming languages together with database query processing. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, page 37–49, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3276945.3276951.

[39] Hanfeng Chen, Alexander Krolik, Bettina Kemme, Clark Verbrugge, and Laurie Hendren. Improving database query performance with automatic fusion. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 63–73, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377555.3377892.

[40] Hanfeng Chen, Alexander Krolik, Erick Lavoie, and Laurie Hendren. Automatic vectorization for MATLAB. In Chen Ding, John Criswell, and Peng Wu, editors, *Languages and Compilers for Parallel Computing*, pages 171–187, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-52709-3_14.

[41] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. Guided region-based GPU scheduling: Utilizing multi-thread parallelism to hide memory latency. In *2013 IEEE 27th International Symposium on Parallel*

and Distributed Processing, pages 441–451, 2013. doi:10.1109/IPDPS.2013.95.

[42] Wei-Yu Chen, Guei-Yuan Lueh, Pratik Ashar, Kaiyu Chen, and Buqi Cheng. Register allocation for Intel processor graphics. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 352–364, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3168806.

[43] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines. *Proc. VLDB Endow.*, 12(5):544–556, jan 2019. doi:10.14778/3303753.3303760.

[44] Periklis Chrysogelos, Panagiotis Sioulas, and Anastasia Ailamaki. Hardware-conscious query processing in GPU-accelerated analytical engines. In *CIDR*, 2019.

[45] Hawon Chu, Seounghyun Kim, Joo-Young Lee, and Young-Kyoon Suh. Empirical evaluation across multiple GPU-accelerated DBMSes. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3399666.3399907.

[46] cloudcores. GitHub - cloudcores/CuAssembler, May 2021. URL: https://github.com/cloudcores/CuAssembler/.

[47] Brett W. Coon, John Erik Lindholm, Peter C. Mills, and John R. Nickolls. Processing an indirect branch instruction in a SIMD architecture, July 2010. URL: https://patents.google.com/patent/US7761697B1/en.

[48] Brett W. Coon, John R. Nickolls, Lars Nyland, Peter C. Mills, and John Erik Lindholm. Indirect function call instructions in a synchronous parallel thread processor, November 2012. URL: https://patents.google.com/patent/US8312254B2/en.

[49] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[50] Gregory Diamos, Benjamin Ashbaugh, Subramaniam Maiyuran, Andrew Kerr, Haicheng Wu, and Sudhakar Yalamanchili. SIMD re-convergence at thread frontiers. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 477–488, 2011. doi:10.1145/2155620.2155676.

[51] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 353–364, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1854273.1854318.

[52] Jesse Doherty and Laurie Hendren. McSAF: A static analysis framework for MATLAB. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, page 132–155, Berlin, Heidelberg, 2012. Springer-Verlag. doi:10.1007/978-3-642-31057-7_7.

[53] Łukasz Domagała, Duco van Amstel, Fabrice Rastello, and P. Sadayappan. Register allocation and promotion through combined instruction scheduling and loop unrolling. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 143–151, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892208.2892219.

[54] Rodrigo Domínguez, Dana Schaa, and David Kaeli. Caracal: Dynamic translation of runtime environments for GPUs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1964179.1964186.

[55] Rodrigo Domínguez and David R. Kaeli. Unstructured control flow in GPGPU. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1194–1202, 2013. doi:10.1109/IPDPSW.2013.247.

[56] Georg Dotzler, Ronald Veldema, and Michael Klemm. JCudaMP: OpenMP/-Java on CUDA. In *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, IWMSE '10, page 10–17, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1808954.1808959.

[57] Michiel du Toit. A C# LIKE implementation that mimics SQL LIKE - CodeProject, Jun 2013. URL: https://www.codeproject.com/Tips/608266/A-Csharp-LIKE-implementation-that-mimics-SQL-LIKE.

[58] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for GPUs: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 1–12, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2254064.2254066.

[59] Niall Emmart, Justin Luitjens, Charles Weems, and Cliff Woolley. Optimizing modular multiplication for NVIDIA's Maxwell GPUs. In *2016 IEEE 23nd Symposium on Computer Arithmetic (ARITH)*, pages 47–54, 2016. doi:10.1109/ARITH.2016.21.

[60] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, 29:33–59, jan 2020. doi:10.1007/s00778-019-00581-w.

[61] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1–2):670–680, sep 2010. doi:10.14778/1920841.1920927.

[62] David Farrell. GitHub - nosferalatu/SimpleGPUHashTable, Mar 2020. URL: https://github.com/nosferalatu/SimpleGPUHashTable.

[63] Sofoklis Floratos, Mengbai Xiao, Hao Wang, Chengxin Guo, Yuan Yuan, Rubao Lee, and Xiaodong Zhang. NestGPU: Nested query processing on GPU. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 1008–1019, 2021. doi:10.1109/ICDE51399.2021.00092.

[64] freedesktop.org. Mesa / mesa · GitLab, Feb 2022. URL: https://gitlab.freedesktop.org/mesa/mesa.

[65] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-in-time GPU compilation for interpreted languages with partial evaluation. *SIGPLAN Not.*, 52(7):60–73, apr 2017. doi:10.1145/3140607.3050761.

[66] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient SIMT control flow. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 25–36, 2011. doi:10.1109/HPCA.2011.5749714.

[67] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, page 407–420, USA, 2007. IEEE Computer Society. doi:10.1109/MICRO.2007.12.

[68] Henning Funke, Sebastian Breß, Stefan Noll, Volker Markl, and Jens Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1603–1618, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183713.3183734.

[69] Henning Funke, Jan Mühlig, and Jens Teubner. Efficient generation of machine code for query compilers. In *Proceedings of the 16th International Workshop on*

Data Management on New Hardware, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3399666.3399925.

[70] Henning Funke and Jens Teubner. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.*, 13(6):884–897, feb 2020. doi:10.14778/3380750.3380758.

[71] Henning Funke and Jens Teubner. Low-latency compilation of SQL queries to machine code. *Proc. VLDB Endow.*, 14(12):2691–2694, jul 2021. doi:10.14778/3476311.3476321.

[72] Rahul Garg and José Nelson Amaral. Compiling Python to a hybrid execution environment. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 19–30, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1735688.1735695.

[73] Rahul Garg and Laurie Hendren. Just-in-time shape inference for array-based languages. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 50–55, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2627373.2627382.

[74] Rahul Garg, Sameer Jagdale, and Laurie Hendren. Velociraptor: A compiler toolkit for array-based languages targeting CPUs and GPUs. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, page 19–24, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2774959.2774967.

[75] Philip B. Gibbons and Steven S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, page 11–16, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/12276.13312.

[76] Mike Giles. Lecture notes in CUDA programming, July 2019. URL: https://people.maths.ox.ac.uk/gilesm/cuda/.

[77] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous APIs: An idiomatic approach. *SIGPLAN Not.*, 53(2):139–153, mar 2018. doi:10.1145/3296957.3173182.

[78] Xiang Gong, Zhongliang Chen, Amir Kavyan Ziabari, Rafael Ubal, and David Kaeli. TwinKernels: An execution model to improve GPU hardware scheduling at compile time. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 39–49, 2017. doi:10.1109/CGO.2017.7863727.

[79] Xun Gong, Xiang Gong, Leiming Yu, and David Kaeli. HAWS: Accelerating GPU wavefront execution through selective out-of-order execution. *ACM Trans. Archit. Code Optim.*, 16(2), April 2019. doi:10.1145/3291050.

[80] J. R. Goodman and W.-C. Hsu. Code scheduling and register allocation in large basic blocks. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, page 88–98, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/2591635.2667158.

[81] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 325–336, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1142473.1142511.

[82] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 215–226, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1007568.1007594.

[83] Scott Gray. GitHub - NervanaSystems/maxas, Jun 2016. URL: https://github.com/NervanaSystems/maxas.

[84] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144, 2011. doi:10.1109/ISPASS.2011.5762730.

[85] Jayanth Gummaraju, Laurent Morichetti, Michael Houston, Ben Sander, Benedict R. Gaster, and Bixia Zheng. Twin Peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 205–216, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1854273.1854302.

[86] Tianyi David Han and Tarek S. Abdelrahman. hiCUDA: High-level GPGPU programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, 2011. doi:10.1109/TPDS.2010.62.

[87] Mark Harris. How to optimize data transfers in CUDA C/C++ | NVIDIA developer blog, Dec 2012. URL: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[88] Mark Harris. How to overlap data transfers in CUDA C/C++ | NVIDIA technical blog, Dec 2012. URL: https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/.

[89] Mark Harris. Unified memory for CUDA beginners | NVIDIA technical blog, Jun 2017. URL: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/.

[90] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. Decoding CUDA binary. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 229–241. IEEE Press, 2019. doi:10.1109/CGO.2019.8661186.

[91] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4), December 2009. doi:10.1145/1620585.1620588.

[92] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 511–524, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1376616.1376670.

[93] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, feb 2011. doi:10.14778/1952376.1952381.

[94] Jiong He, Mian Lu, and Bingsheng He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *Proc. VLDB Endow.*, 6(10):889–900, aug 2013. doi:10.14778/2536206.2536216.

[95] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled CPU-GPU architectures. *Proc. VLDB Endow.*, 8(4):329–340, dec 2014. doi:10.14778/2735496.2735497.

[96] Max Heimel and Volker Markl. A first step towards GPU-assisted query optimization. In Rajesh Bordawekar and Christian A. Lang, editors, *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012, Istanbul, Turkey, August 27, 2012*, pages 33–44, 2012. URL: http://www.adms-conf.org/heimel_adms12.pdf.

[97] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, jul 2013. doi:10.14778/2536360.2536370.

[98] Laurie Hendren. Lecture notes in COMP 621: Program analysis and transformations, September 2015. URL: http://www.sable.mcgill.ca/~hendren/621/.

[99] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 381–392, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1950365.1950409.

[100] Joseph Huber, Melanie Cornelius, Giorgis Georgakoudis, Shilei Tian, Jose M Monsalve Diaz, Kuter Dinel, Barbara Chapman, and Johannes Doerfert. Efficient execution of openmp on gpus. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 41–52, 2022. doi:10.1109/CGO53902.2022.9741290.

[101] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[102] IEEE and The Open Group. The Base Specifications Issue 7, 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/mindex.html.

[103] BlazingSQL Inc. BlazingSQL | high performance SQL engine on RAPIDS AI, 2022. URL: https://blazingsql.com/.

[104] James A. Jablin, Thomas B. Jablin, Onur Mutlu, and Maurice Herlihy. Warp-aware trace scheduling for GPUs. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 163–174, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628071.2628101.

[105] Dejice Jacob, Phil Trinder, and Jeremy Singer. Python programmers have GPUs too: Automatic Python loop parallelization with staged dependence analysis. In

Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS 2019, page 42–54, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359619.3359743.

[106] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Scarpazza. Dissecting the NVidia Turing T4 GPU via microbenchmarking, 03 2019. URL: https://arxiv.org/abs/1903.07486.

[107] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. Dissecting the NVIDIA Volta GPU architecture via microbenchmarking, 04 2018. URL: http://arxiv.org/abs/1804.06826.

[108] Robert Ohannessian Jr, Michael Alan Fetterman, Olivier Giroux, Jack H. Choquette, Xiaogang Qiu, Shirish Gadre, and Meenaradchagan Vishnu. System, method, and computer program product for implementing software-based scoreboarding, August 2015. URL: https://patents.google.com/patent/US20150220341A1/en.

[109] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, page 55–62, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2236584.2236592.

[110] Charu Kalra. Design and evaluation of register allocation on GPUs. Master's thesis, Northeastern University, 2015.

[111] T. Karnagel, Dirk Habich, and Wolfgang Lehner. Local vs. global optimization: Operator placement strategies in heterogeneous environments. *CEUR Workshop Proceedings*, 1330:48–55, 01 2015.

[112] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10(7):733–744, mar 2017. doi:10.14778/3067421.3067423.

[113] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. Heterogeneity-aware operator placement in column-store DBMS. *Datenbank-Spektrum*, 14:211–221, 11 2014. doi:10.1007/s13222-014-0167-9.

[114] Tomas Karnagel, René Müller, and Guy M. Lohman. Optimizing GPU-accelerated group-by and aggregation. In *ADMS@VLDB*, pages 13–24, 2015.

[115] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. Fast queries over heterogeneous data through engine customization. *Proc. VLDB Endow.*, 9(12):972–983, aug 2016. doi:10.14778/2994509.2994516.

[116] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, sep 2018. doi:10.14778/3275366.3284966.

[117] Timo Kersten, Viktor Leis, and Thomas Neumann. Tidy Tuples and Flying Start: Fast compilation and fast execution of relational queries in Umbra. *VLDB J.*, 30:883–905, 2021.

[118] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. doi:10.1109/ISCA45697.2020.00047.

[119] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013. doi:10.1145/2400682.2400690.

[120] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Comput.*, 38(3):157–174, mar 2012. doi:10.1016/j.parco.2011.09.001.

[121] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. Building efficient query engines in a high-level language. *Proc. VLDB Endow.*, 7(10):853–864, jun 2014. doi:10.14778/2732951.2732959.

[122] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208, 2018. doi:10.1109/ICDE.2018.00027.

[123] Marcelina Kościelnicka, Ben Skeggs, Martin Peres, Maarten Lankhorst, Roy Spliet, Christoph Bumiller, Marcin Ślusarz, Emil Velikov, and Francisco Jerez. GitHub - envytools/envytools, May 2021. URL: https://github.com/envytools/envytools.

[124] Mayank Kothiya. Understanding the ISA impact on GPU architecture. Master's thesis, North Carolina State University, 2014.

[125] Alexander Krolik, Clark Verbrugge, and Laurie Hendren. r3d3: Optimized query compilation on GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 277–288, 2021. doi:10.1109/CGO51591.2021.9370323.

[126] Serge Lamikhov-Center. GitHub - serge1/ELFIO, Sep 2020. URL: https://github.com/serge1/ELFIO.

[127] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society. doi:10.1109/CGO.2004.1281665.

[128] Andrew S. D. Lee and Tarek S. Abdelrahman. Launch-time optimization of OpenCL GPU kernels. In *Proceedings of the General Purpose GPUs*, GPGPU-10, page 32–41, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3038228.3038236.

[129] Rubao Lee, Minghong Zhou, Chi Li, Shenggang Hu, Jianping Teng, Dongyang Li, and Xiaodong Zhang. The art of balance: A RateupDB™ experience of building a CPU/GPU hybrid database product. *Proc. VLDB Endow.*, 14(12):2999–3013, jul 2021. doi:10.14778/3476311.3476378.

[130] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, page 1–11, USA, 2010. IEEE Computer Society. doi:10.1109/SC.2010.36.

[131] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1504176.1504194.

[132] Seyong Lee and Jeffrey S. Vetter. OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 115–120, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2600212.2600704.

[133] Martin Leitner-Ankerl. GitHub - martinus/robin-hood-hashing, May 2021. URL: https://github.com/martinus/robin-hood-hashing.

[134] Alan Leung, Ondřej Lhoták, and Ghulam Lashari. Automatic parallelization for graphics processing units. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, page 91–100, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1596655.1596670.

[135] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. HippogriffDB: Balancing I/O and gpu bandwidth in big data analytics. *Proc. VLDB Endow.*, 9(14):1647–1658, oct 2016. doi:10.14778/3007328.3007331.

[136] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March 2008. doi:10.1109/MM.2008.31.

[137] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee. *C-for-Metal: High Performance SIMD Programming on Intel GPUs*, page 289–300. IEEE Press, 2021. URL: https://doi.org/10.1109/CGO51591.2021.9370324.

[138] Justin Luitjens. Faster parallel reductions on Kepler | NVIDIA developer blog, Feb 2012. URL: https://developer.nvidia.com/blog/faster-parallel-reductions-kepler/.

[139] Mark Harris Luke Durant, Olivier Giroux and Nick Stam. Inside Volta: The world's most advanced data center GPU | NVIDIA developer blog, May 2017. URL: https://developer.nvidia.com/blog/inside-volta/.

[140] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Pump up the volume: Processing large data on GPUs with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1633–1649, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389705.

[141] Marcello Maggioni and Charu Chandrasekaran. Apple LLVM GPU compiler: Embedded dragons, 2017. US LLVM Developers' Meeting.

[142] Hidehiko Masuhara and Yusuke Nishiguchi. A data-parallel extension to Ruby for GPGPU: Toward a framework for implementing domain-specific optimizations. In *Proceedings of the 9th ECOOP Workshop on Reflection, AOP, and*

Meta-Data for Software Evolution, RAM-SE '12, page 3–6, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2237887.2237888.

[143] Kothiya Mayank, Hongwen Dai, Jizeng Wei, and Huiyang Zhou. Analyzing graphics processor unit (GPU) instruction set architectures. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 155–156, 2015. doi:10.1109/ISPASS.2015.7095794.

[144] Suejb Memeti and Sabri Pllana. HSTREAM: A directive-based language extension for heterogeneous stream computing. In *2018 IEEE International Conference on Computational Science and Engineering (CSE)*, pages 138–145, 2018. doi:10.1109/CSE.2018.00026.

[145] Gleison Mendonça, Breno Guimarães, Péricles Alves, Márcio Pereira, Guido Araújo, and Fernando Magno Quintão Pereira. DawnCC: Automatic annotation for data parallelism and offloading. *ACM Trans. Archit. Code Optim.*, 14(2), may 2017. doi:10.1145/3084540.

[146] Sina Meraji, Berni Schiefer, Lan Pham, Lee Chu, Peter Kokosielis, Adam Storm, Wayne Young, Chang Ge, Geoffrey Ng, and Kajan Kanagaratnam. Towards a hybrid design for fast query processing in DB2 with BLU acceleration using graphical processing units: A technology demonstration. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1951–1960, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2903735.

[147] Duane Merrill and Michael Garland. Single-pass parallel prefix scan with decoupled lookback. Technical report, NVIDIA, 2016.

[148] Nicholas Moore, Miriam Leeser, and Laurie Smith King. Kernel specialization for improved adaptability and performance on graphics processing units (GPUs). In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1037–1048, 2013. doi:10.1109/IPDPS.2013.31.

[149] John Magnus Morton, Kuba Kaszyk, Lu Li, Jiawen Sun, Christophe Dubach, Michel Steuwer, Murray Cole, and Michael F. P. O'Boyle. DelayRepay: Delayed execution for kernel fusion in Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2020, page 43–56, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3426422.3426980.

[150] Todd Mostak. An overview of MapD (Massively Parallel Database), Sep 2014. URL: http://www.smallake.kr/wp-content/uploads/2014/09/mapd_overview.pdf.

[151] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[152] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011. doi:10.14778/2002938.2002940.

[153] Thomas Neumann. Database architects: Linear time liveness analysis, Apr 2020. URL: http://databasearchitects.blogspot.com/2020/04/linear-time-liveness-analysis.html.

[154] John R. Nickolls, Richard Craig Johnson, Robert Steven Glanville, and Guillermo Juan Rozas. Unanimous branch instructions in a parallel thread processor, March 2011. URL: https://patents.google.com/patent/US20110072248/en.

[155] Gabriel Noaje, Christophe Jaillet, and Michaël Krajecki. Source-to-source code translator: OpenMP C to CUDA. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications*, HPCC '11, page 512–519, USA, 2011. IEEE Computer Society. doi:10.1109/HPCC.2011.73.

[156] Cedric Nugteren and Henk Corporaal. Introducing 'Bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the*

5th Annual Workshop on General Purpose Processing with Graphics Processing Units, GPGPU-5, page 1–10, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2159430.2159431.

[157] Cedric Nugteren and Henk Corporaal. Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs. *ACM Trans. Archit. Code Optim.*, 11(4), dec 2014. doi:10.1145/2665079.

[158] NVIDIA. GeForce GTX 1080 whitepaper, 2016. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

[159] NVIDIA. Driver persistence :: GPU deployment and management documentation, Jun 2020. URL: https://docs.nvidia.com/deploy/driver-persistence/index.html.

[160] NVIDIA. NVIDIA Ampere GA102 GPU architecture, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf.

[161] NVIDIA. Best practices guide :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[162] NVIDIA. CUDA binary utilities :: CUDA Toolkit documentation, Aug 2021. URL: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

[163] NVIDIA. CUDA driver API :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/cuda-driver-api/index.html.

[164] NVIDIA. CUDA occupancy calculator :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html.

[165] NVIDIA. CUDA runtime API :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html.

[166] NVIDIA. libdevice user's guide :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/libdevice-users-guide/index.html.

[167] NVIDIA. NVCC :: CUDA Toolkit documentation, Oct 2021. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[168] NVIDIA. Programming guide :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[169] NVIDIA. PTX compiler APIs :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/ptx-compiler-api/index.html.

[170] NVIDIA. PTX ISA :: CUDA Toolkit documentation, Aug 2021. URL: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[171] NVIDIA. Volta tuning guide :: CUDA Toolkit documentation, Nov 2021. URL: https://docs.nvidia.com/cuda/volta-tuning-guide/index.html.

[172] NVIDIA. CUB: Main page, Jan 2022. URL: https://nvlabs.github.io/cub/.

[173] NVIDIA. NVVM IR :: CUDA Toolkit documentation, Jan 2022. URL: https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.

[174] NVIDIA. Thrust :: CUDA Toolkit documentation, Jan 2022. URL: https://docs.nvidia.com/cuda/thrust/index.html.

[175] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-compatible library for NVIDIA GPU calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in*

The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS), 2017. URL: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[176] openacc standard.org. OpenACC programming and best practices guide, May 2021. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf.

[177] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving execution efficiency of just-in-time compilation based query processing on GPUs. *Proc. VLDB Endow.*, 14(2):202–214, October 2020. doi:10.14778/3425879.3425890.

[178] Johns Paul, Jiong He, and Bingsheng He. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1935–1950, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2882903.2915224.

[179] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. *MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures*, page 1413–1425. Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3448016.3457254.

[180] Fernando Magno Quintão Pereira. Lecture notes in DCC888: Program analysis and optimization (worklist algorithms), January 2020. URL: https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/WorkList.pdf.

[181] Shlomit S. Pinter. Register allocation with instruction scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 248–257, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/155090.155114.

[182] Holger Pirk, Stefan Manegold, and Martin Kersten. Waste not... efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 508–519, 2014. doi:10.1109/ICDE.2014.6816677.

[183] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, October 2016. doi:10.14778/3007328.3007336.

[184] pocl developers. GitHub - pocl/pocl, Jan 2022. URL: https://github.com/pocl/pocl/.

[185] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895–913, September 1999. doi:10.1145/330249.330250.

[186] LLVM Project. User guide for NVPTX back-end — LLVM 13 documentation, Jan 2022. URL: https://llvm.org/docs/NVPTXUsage.html.

[187] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 242–253, 2019. doi:10.1109/CGO.2019.8661176.

[188] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3rd edition, 2002.

[189] Syed Mohammad Aunn Raza, Periklis Chrysogelos, Panagiotis Sioulas, Vladimir Indjic, Angelos Christos Anadiotis, and Anastasia Ailamaki. GPU-accelerated data management under the test of time. In *Online proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*, page 11, 2020. This article is published under a Creative Commons Attribution License 3.0. URL: http://infoscience.epfl.ch/record/277001.

[190] Nico Reissmann, Thomas L. Falch, Benjamin A. Bjørnseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre. Efficient control flow restructuring

for GPUs. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 48–57, 2016. doi:10.1109/HPCSim.2016.7568315.

[191] Viktor Rosenfeld, Sebastian Breß, and Volker Markl. Query processing on heterogeneous CPU/GPU systems. *ACM Comput. Surv.*, 55(1), jan 2022. doi:10.1145/3485126.

[192] Viktor Rosenfeld, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. Performance analysis and automatic tuning of hash aggregation on GPUs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3329785.3329922.

[193] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: A compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 49–68, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2522715.

[194] Eyal Rozenberg and Peter Boncz. Faster across the PCIe bus: A GPU library for lightweight decompression: Including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3076113.3076122.

[195] Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha. Parakeet: A just-in-time parallel accelerator for Python. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar'12, page 14, USA, 2012. USENIX Association.

[196] Ran Rui, Hao Li, and Yi-Cheng Tu. Join algorithms on GPUs: A revisit after seven years. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 2541–2550, 2015. doi:10.1109/BigData.2015.7364051.

[197] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-GPU environment. *Proc. VLDB Endow.*, 14(4):708–720, dec 2020. doi:10.14778/3436905.3436927.

[198] Ran Rui and Yi-Cheng Tu. Fast equi-join algorithms on GPUs: Design and implementation. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, SSDBM '17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3085504.3085521.

[199] Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Scott Mahlke. Paragon: Collaborative speculative loop execution on GPU and CPU. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, page 64–73, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2159430.2159438.

[200] Shigeyuki Sato and Hideya Iwasaki. A skeletal parallel framework with fusion optimizer for GPGPU programming. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, page 79–94, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-10672-9_8.

[201] Charitha Saumya, Kirshanthan Sundararajah, and Milind Kulkarni. Darm: Control-flow melding for simt thread divergence reduction. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–13, 2022. doi:10.1109/CGO53902.2022.9741285.

[202] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1617–1632, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3380595.

[203] Ghassan Shobaki, Justin Bassett, Mark Heffernan, and Austin Kerbow. Graph transformations for register-pressure-aware instruction scheduling. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC 2022, page 41–53, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3497776.3517771.

[204] Ghassan Shobaki, Austin Kerbow, and Stanislav Mekhanoshin. Optimizing occupancy and ILP on the GPU using a combinatorial approach. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, page 133–144, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368826.3377918.

[205] Dhirendra Singh, Ishan Joshi, and Jaytrilok Choudhary. Survey of GPU based sorting algorithms. *International Journal of Parallel Programming*, 46, 04 2017. doi:10.1007/s10766-017-0502-5.

[206] P. Sioulas, P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hardware-conscious hash-joins on GPUs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 698–709, 2019. doi:10.1109/ICDE.2019.00068.

[207] Evangelia A. Sitaridi and Kenneth A. Ross. Optimizing select conditions on GPUs. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485278.2485282.

[208] Evangelia A. Sitaridi and Kenneth A. Ross. GPU-accelerated string matching for database applications. *The VLDB Journal*, 25(5):719–740, oct 2016. doi:10.1007/s00778-015-0409-y.

[209] Rafael Sotomayor, Luis Miguel Sanchez, Javier Garcia Blas, Javier Fernandez, and J. Daniel Garcia. Automatic CPU/GPU generation of multi-versioned OpenCL kernels for C++ scientific applications. *Int. J. Parallel Program.*, 45(2):262–282, apr 2017. doi:10.1007/s10766-016-0425-6.

[210] Matthias Springer and Hidehiko Masuhara. Object support in an array-based GPGPU extension for Ruby. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, page 25–31, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2935323.2935327.

[211] Matthias Springer, Peter Wauligmann, and Hidehiko Masuhara. Modular array-based GPU computing in a dynamically-typed language. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2017, page 48–55, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3091966.3091974.

[212] M. Steuwer, T. Remmelg, and C. Dubach. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85, 2017. doi:10.1109/CGO.2017.7863730.

[213] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, page 205–217, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2784731.2784754.

[214] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-Store: A column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 553–564. VLDB Endowment, 2005.

[215] Alex Suhan. Massive throughput database queries with LLVM on GPUs, Apr 2016. URL: https://www.omnisci.com/blog/massive-throughput-database-queries-with-llvm-on-gpus.

[216] Dean Takahashi. Nintendo Switch specs: less powerful than PlayStation 4, Dec 2016. URL: https://venturebeat.com/2016/12/14/nintendo-switch-specs-less-powerful-than-playstation-4/.

[217] PG-Strom Development Team. Home - PG-Strom manual, Apr 2020. URL: https://heterodb.github.io/pg-strom/.

[218] The Halide team. GitHub - halide/Halide, Jan 2022. URL: https://github.com/halide/Halide/.

[219] Andrew Tolmach. Lecture notes in CS322: Languages and compiler design II, April 2012. URL: https://web.cecs.pdx.edu/~apt/cs322/.

[220] Diego G. Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A. Boncz. Optimizing group-by and aggregation using GPU-CPU co-processing. In *ADMS@VLDB*, 2018.

[221] Transaction Processing Performance Council. TPC Benchmark H, 2017.

[222] Damien Triolet. GP104 : 7.2 milliards de transistors en 16 nm - Nvidia GeForce GTX 1080, le premier GPU 16nm en test ! - HardWare.fr, May 2016. URL: https://www.hardware.fr/articles/948-2/gp104-7-2-milliards-transistors-16-nm.html.

[223] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 335–344, New York, NY, USA, 2012. Association for Computing Machinery. doi:10.1145/2370816.2370865.

[224] Wladimir J. van der Laan. GitHub - laanwj/decuda, Jun 2010. URL: https://github.com/laanwj/decuda.

[225] Clark Verbrugge. Lecture notes on compiler optimization, May 2021. URL: http://www.sable.mcgill.ca/~clump/compileropt/.

[226] Clark Verbrugge, Christopher J. F. Pickett, Alexander Krolik, and Allan Kielstra. Exhaustive analysis of thread-level speculation. In *Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems*, SEPS 2016, page 25–34, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/3002125.3002127.

[227] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, page 191–202. IEEE Press, 2014. doi:10.1109/SC.2014.21.

[228] Fabian Wahlster. Implementing SPMD control flow in LLVM using reconverging CFGs, 2019. European LLVM Developers' Meeting. URL: https://llvm.org/devmtg/2019-04/slides/Poster-Wahlster-Implementing_SPMD_control_flow_in_LLVM_using_reconverging_CFG.pdf.

[229] Kaibo Wang, Kai Zhang, Yuan Yuan, Siyuan Ma, Rubao Lee, Xiaoning Ding, and Xiaodong Zhang. Concurrent analytical query processing with GPUs. *Proc. VLDB Endow.*, 7(11):1011–1022, jul 2014. doi:10.14778/2732967.2732976.

[230] Zheng Wang, Dominik Grewe, and Michael F. P. O'boyle. Automatic and portable mapping of data parallel programs to OpenCL for GPU-based heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 11(4), dec 2014. doi:10.1145/2677036.

[231] Michael Wolfe. Implementing the PGI accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, page 43–50, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1735688.1735697.

[232] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel Weaver: Automatically fusing database primitives for efficient GPU computation. In *2012*

45th Annual IEEE/ACM International Symposium on Microarchitecture, pages 107–118, 2012. doi:10.1109/MICRO.2012.19.

[233] Haicheng Wu, Gregory Diamos, Tim Sheard, Molham Aref, Sean Baxter, Michael Garland, and Sudhakar Yalamanchili. Red Fox: An execution environment for relational query processing on GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 44–54, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2544137.2544166.

[234] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, page 2433–2442, USA, 2012. IEEE Computer Society. doi:10.1109/IPDPSW.2012.300.

[235] Haicheng Wu, Gregory Diamos, Jin Wang, Si Li, and Sudhakar Yalamanchili. Characterization and transformation of unstructured control flow in bulk synchronous GPU applications. *Int. J. High Perform. Comput. Appl.*, 26(2):170–185, may 2012. doi:10.1177/1094342011434814.

[236] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, and Robert Hundt. gpucc: An open-source GPGPU compiler. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 105–116, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2854038.2854041.

[237] Haoran Xu and Fredrik Kjolstad. Copy-and-patch compilation: A fast compilation algorithm for high-level languages and bytecode. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021. doi:10.1145/3485513.

[238] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. Relational joins on GPUs: A closer look. *IEEE Transactions on Parallel and Distributed Systems*, 28(9):2663–2673, 2017. doi:10.1109/TPDS.2017.2677451.

[239] Da Yan, Wei Wang, and Xiaowen Chu. Optimizing batched winograd convolution on GPUs. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*, San Diego, CA, USA, 2020. ACM. doi:10.1145/3332466.3374520.

[240] Da Yan, Wei Wang, and Xiaowen Chu. An llvm-based open-source compiler for nvidia gpus. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 448–449, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3503221.3508428.

[241] Shengen Yan, Guoping Long, and Yunquan Zhang. StreamScan: Fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, page 229–238, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2442516.2442539.

[242] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, page 86–97, New York, NY, USA, 2010. Association for Computing Machinery. doi:10.1145/1806596.1806606.

[243] Yi-Ping You and Szu-Chieh Chen. Vector-aware register allocation for GPU shader processors. In *2015 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pages 99–108, 2015. doi:10.1109/CASES.2015.7324550.

[244] Yulong Yu, Weijun Xiao, Xubin He, He Guo, Yuxin Wang, and Xin Chen. A stall-aware warp scheduling for dynamically optimizing thread-level parallelism in GPGPUs. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 15–24, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2751205.2751234.

[245] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The yin and yang of processing data warehousing queries on GPU devices. *Proc. VLDB Endow.*, 6(10):817–828, aug 2013. doi:10.14778/2536206.2536210.

[246] Hou Yunqing. GitHub - hyqneuron/asfermi, Mar 2015. URL: https://github.com/hyqneuron/asfermi/.

[247] yuzu emulator team. GitHub - yuzu-emu/yuzu: Nintendo Switch Emulator, Feb 2022. URL: https://github.com/yuzu-emu/yuzu.

[248] F. Zhang and E.H. D'Hollander. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering*, 30(4):231–245, 2004. doi:10.1109/TSE.2004.1274043.

[249] Kai Zhang, Feng Chen, Xiaoning Ding, Yin Huai, Rubao Lee, Tian Luo, Kaibo Wang, Yuan Yuan, and Xiaodong Zhang. Hetero-DB: Next generation high-performance database systems by best utilizing heterogeneous computing and storage resources. *J. Comput. Sci. Technol.*, 30(4):657–678, 2015. doi:10.1007/s11390-015-1553-y.

[250] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proc. VLDB Endow.*, 6(12):1374–1377, August 2013. doi:10.14778/2536274.2536319.

[251] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. Understanding the GPU microarchitecture to achieve bare-metal performance tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium*

on Principles and Practice of Parallel Programming, PPoPP '17, page 31–43, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3018743.3018755.