Give a JIT on GPUs: NVRTC for Code-Generating Database Systems

Anton Sachnov
University of Bamberg
anton.sachnov@uni-bamberg.de

Leonard von Merzljak
TUM
leonard.von-merzljak@tum.de

Maximilian E. Schüle
University of Bamberg
maximilian.schuele@uni-bamberg.de

***Abstract*—Code generation for GPU database systems creates fast runtime code but suffers from expensive compilation and requires all modules to be compiled before execution. This paper investigates how the NVIDIA runtime compiler (NVRTC) accelerates compilation for code-generating GPU database systems. Using NVRTC, we demonstrate just-in-time (JIT) compilation for a tuple-at-a-time (push-based) query execution model. For an exemplary analytical query, NVRTC accelerates the compilation time by a a factor of eight compared to NVIDIA's CUDA compiler (nvcc). This will allow the development of code-generating GPU database systems for adaptive query execution.**

I. Introduction

Interpretation of query plans—the traditional Volcano-based execution model [1]—suffers from one function call per operator and tuple. The push-based execution model [2], which inverses the data flow, together with code generation eliminates any interpreted function call at execution time but requires one function call per operator at compile time. As the generation of C++ code is time-consuming, HyPer [3], [4] is the first code-generating database system that uses low-level virtual machine (LLVM) [5] assembly for faster compilation times. This allows for a just-in-time (JIT) [6], [7] compilation of user-defined SQL queries.

HyPer supports concurrent execution on multiple CPUs through morsel-driven parallelism but lacks support for data-level parallelism. The *single instruction, multiple threads* (SIMT) execution model, on which graphics processing units (GPUs) rely, provides data-level parallelism. Therefore, GPU database systems [8], [9] have been developed that provide a high throughput by performing the same instruction on multiple tuples in parallel [10], [11]. Their development relies on general-purpose computing on GPUs (GPGPU) based upon platforms such as NVIDIA's Compute Unified Device Architecture (CUDA).

NVIDIA's CUDA architecture expects instructions for its parallel thread execution (PTX) virtual machine as input. The graphics driver then translates the PTX instructions (in ASCII representation) into binary code to run on the GPU. When porting code generation to the GPU, one can use NVIDIA's compiler `nvcc` to generate CUDA C/C++ code with comparable compilation times as GNU gcc or let LLVM address the GPU. LLVM uses an intermediate representation (IR) that is mapped to the device-specific assembly. When addressing the GPU, LLVM maps the IR to the PTX instruction set. This constitutes another indirection to a virtual machine, which could be addressed directly. To avoid any indirection for JIT compilation on CPUs, asmjit[1] directly creates assembly code for x86, x86_64, and AArch64 architectures. For GPUs, the NVIDIA runtime compiler (NVRTC), introduced with CUDA version 7.0 in 2014 [12], generates PTX code from a C++ string at runtime (cf. Figure 1).

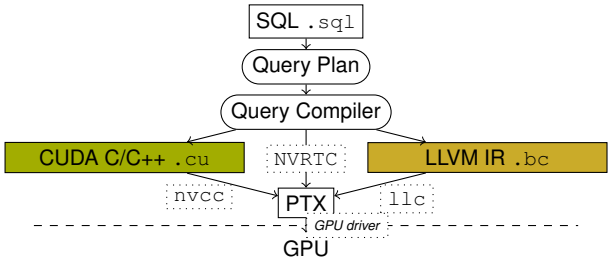

Fig. 1. Code-generation for GPU: The system creates a query plan based on SQL as input. The query compiler generates the code—either as CUDA C/C++ or as LLVM IR—to be executed on the GPU, which is then compiled to PTX using the `nvcc` or the `llc` compiler. Instead, NVRTC allows the query compiler to generate and execute the PTX code directly.

We argue that NVRTC is well suited for code-generating GPU database systems [13]–[16] with faster compilation times than `nvcc`. To demonstrate query-compilation to GPU, we implement an analytical query based on a tuple-at-a-time push-based query execution model and measure its compilation and execution time.

Tuple-at-a-time execution models without code generation are unsuited for GPU database systems as each interpreted call would trigger memory transfer to the GPU and the call of a GPU kernel. Therefore, operator-at-a-time execution models, that first transfer the data block-wise and then execute a relational operator, amortise the overhead for launching a GPU kernel. However, HetExchange [17] proved that code-generation enables tuple-at-a-time query execution for GPU database systems and introduced a pack operators for data exchange to/from GPU. In this paper, we demonstrate code generation for GPUs based on a push-based tuple-at-a-time execution model and compare the compilation time with nvcc to NVRTC.

The paper is structured as follows: Section II explains the background on NVRTC for its embedding within code-

[1]https://asmjit.com/

generating GPU database systems based on HetExchange. Section III presents the interaction of NVRTC with a tuple-at-a-time (push-based) execution model suited for code-generation to GPU. Section IV compares the compilation time for code-generation with NVRTC to the one with nvcc and presents the execution and compilation times of a selected query on the GPU. Section V concludes this paper with an outlook on future database architectures based on NVRTC.

II. Background: NVRTC and HetExchange

NVRTC generates CUDA C/C++ code at runtime[2]. Without NVRTC, users had to compile their CUDA C/C++ before execution, which required to spawn an additional process. To avoid an additional process, NVRTC compiles PTX code from a character string at runtime and returns a handle to the generated code. This dynamic compilation facility allows for optimizations and performance improvements that are not achievable through static compilation offline. The *nvJitLink* library or the *cuLinkAddData* function from the CUDA Driver API then loads and links the generated code with other modules.

HetExchange [17] is a query parallelization technique that extends the Volcano exchange operator. Like the original exchange operator, HetExchange relies on asynchronous queues to parallelise sequential operators. Thereby, HetExchange provides vertical, horizontal, and bushy parallelism. HetExchange's addition to the exchange operator is the ability to parallelise query plans across CPUs and GPUs. As a result, it is capable of parallelising a single query plan across multiple CPUs, multiple GPUs, or even both CPUs and GPUs in a co-processing fashion.

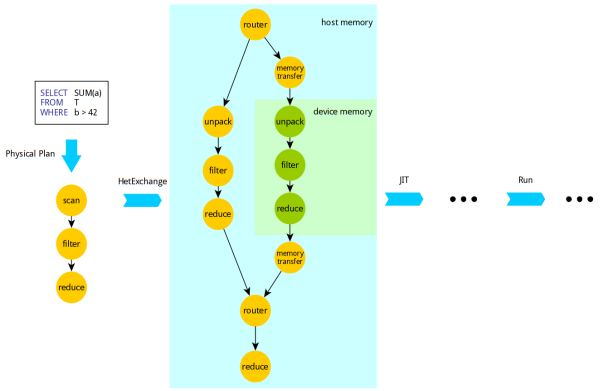

Fig. 2. Simplified visualisation of the lifetime of a query in HetExchange.

To illustrate the workings of HetExchange, we want to discuss parallelisation of a simple query across two cores of a CPU and one GPU. Figure 2 shows a simple query plan consisting of three operators: a table-scan, followed by a filter, and a subsequent aggregation. To parallelise processing, HetExchange incrementally adds operators to this query plan which encapsulate the required control and data flow.

[2]https://docs.nvidia.com/cuda/nvrtc/index.html

```
1 create table products(p_id int primary key, category int,
      price int);
2 create table sellers(s_id int primary key, region int);
3 create table sales(product_id int, seller_id int);
4
5 select category, sum(price)
6 from sales, sellers, products
7 where product_id = p_id and seller_id=s_id and region=0
8 group by category
```

Listing 1. Exemplary query supported by our NVRTC prototype.

The router operator added by HetExchange is responsible for controlling the degree of parallelism. Conceptually, it fulfils the same tasks as the exchange operator in the Volcano system. The router operator instantiates one pipeline of filter and aggregation operators to run on the GPU and another two pipelines of filter and aggregation operators to run on two separate CPU cores. In general, the router operator would instantiate one CPU-pipeline for each CPU hardware thread and one GPU pipeline for each GPU. For GPU-pipelines, HetExchange additionally adds cpu2gpu and gpu2cpu operators, which are responsible for starting a kernel running the pipeline on the GPU and then wait for the kernel to terminate and thereby transfer control back to the CPU. Since the CPU and the GPU have separate memory spaces, HetExchange also has to encapsulate the data flow to make sure that, e.g., a pipeline running in a kernel on the GPU can access its input data even though it originally only resided in CPU-RAM. For this purpose, HetExchange adds a mem-move operator that transfers memory from the CPU to the GPU or vice versa. HetExchange must try to minimise their costs, which is achieved by batching tuples and transferring them as a block.

The router operator then sends those chunks to either the GPU pipeline or to one of the CPU pipelines. The mem-move operator transfers chunks of data from CPU-RAM to GPU-RAM to make them accessible by the GPU. Also, the mem-move operator transfers the intermediate result of the aggregation from the GPU back to the CPU so that the CPU can combine all intermediate results into a final result. Finally, HetExchange adds an unpack operator. Since HetExchange uses the push model but transfers data in chunks, the unpack operator is needed to start an iteration over a chunk.

HetExchange's system architecture is integrated to Proteus, an LLVM-based code-generating and analytical database system. Using its LLVM backend to generate PTX code, HetExchanges addresses NVIDIA GPUs. Instead of LLVM, we are generating PTX code directly using nvcc and NVRTC.

III. Conception

The push-based query execution model follows the conception of HetExchange. Based on HetExchange, we create a code-generation framework using NVRTC. This section describes the current prototype, which is capable of running the query depicted in Listing 1. Figure 3 visualises the corresponding query plan on a star schema with one fact table (*sales*) and two dimension tables. Listing 2 presents the generated GPU kernel with colour-coding to map the corresponding relational

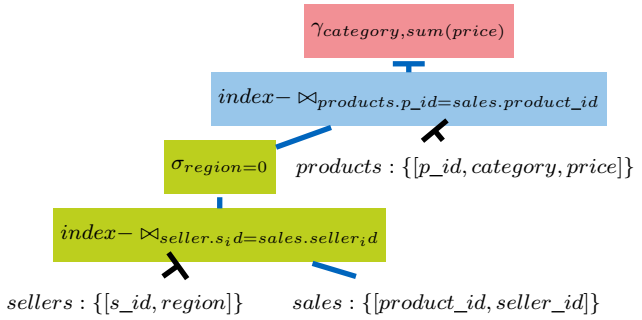

Fig. 3. Query plan to Listing 1 composed of an aggregation (red), an index-join (blue) and once combined with a selection (green).

operators to the generated code lines. Beyond table scan, the prototype supports selection, projection, aggregation and index join.

A. Table Scan and Selection

Having transferred the data to the device memory, we can start our pipeline on the GPU. For the table scan on the GPU—instead of a for-loop to process a morsel on the CPU—each GPU thread processes one tuple. The tuple is determined by the thread identifier (line 5), whose number should be less than the chunk size `n` (line 6). Since we use the push model, we implement selections as a tuple-at-a-time if-expression, resulting in a single statement (line 7).

B. Index-Join

Because we store dimension tables as an array sorted on the primary key and require the primary key to be an ascending number starting from zero with no gaps, we can implement joins as a simple, fast array lookup, where the foreign key in the fact table denotes the offset of the join tuple in the dimension relation. Referring to our example, the first join of *sales* with *sellers* is condensed into an index look-up for the aggregation (line 7), the second join with *products* results in an index look-up in line 8.

C. Aggregation

For aggregations (i.e., summation), we use a lock-free hash-table based upon open addressing and linear probing[3]. We use atomic operations to update the aggregated values. Implementing a sum over an array is more complicated on the GPU because if we use one CUDA thread to process one element, we have to use expensive atomic operations for each element to update the global sum. We can decrease the number of atomic operations by a factor of 32 with the help of warp-aggregated atomics[4]. Fortunately, due to compiler improvements, the NVIDIA compiler can automatically transform code to use warp-aggregated atomics. We experimentally verified that explicitly using warp-aggregated atomics, therefore, does not bring performance improvements over using atomic operations for each element.

Despite automatically using warp-aggregated atomics, the kernel is still very slow. To increase throughput, we have to significantly reduce the number of atomic operations. We can achieve this by adjusting the execution configuration: instead of starting one CUDA thread for each element, we start many fewer threads but let each thread process more elements. A thread computes the sum in thread-local memory and only atomically updates the global sum after it finished processing all elements assigned to it. This allows us to arbitrarily reduce the number of atomic operations.

```
1  extern "C" __global__ void kernel(
2   Sale *begin, unsigned int n, Product *d_products,
3   Seller *d_sellers, int capacity,
4   HTEntry *device_entries, unsigned int unused_key) {
5   auto i = blockIdx.x * blockDim.x + threadIdx.x;
6   if (i < n) {
7    if (d_sellers[begin[i].seller_id].seller_region==0u){
8     auto &product = d_products[begin[i].product_id];
9     int key = product.product_category;
10    int value = product.product_price;
11    int slot = get_hash(key) & (capacity - 1);
12    while (true) {
13     int old = atomicCAS(&device_entries[slot].key,
            unused_key, key);
14     if (old == unused_key  old == key) {
15      atomicAdd(&device_entries[slot].value, value);
16      break;
17     }
18     slot = (slot + 1) & (capacity - 1);
19  }}}}
```

Listing 2. Generated GPU kernel using index-join (blue), once combined with a selection (green) and the hash-table for aggregation (red).

IV. Evaluation

The goal of this subsection is to evaluate the performance of a complex query (Listing 1) parallelised across our CPU and our GPU. The star schema consists of three relations. The sales relation is 3 GB in size and is the fact table. The products and sellers relations are the dimension tables and we will vary their sizes between 3 MB and 20 MB each to illustrate caching behaviour. The query computes the sales revenue per product category in a specific seller's region. First, we join the sales and sellers relation, then we filter by the seller's region, join with the products relation, and finally, group by product category to aggregate all product prices.

System: Intel(R) Xeon(R) W-2295 CPU (18 cores @ 3.00GHz, hyper-threading), 128 GB DDR4 RAM, NVIDIA RTX A2000 12 GB.

Software: Ubuntu 24.04, gcc v13.2.0, nvcc v12.0.140.

A. Compilation Time

Table I shows the compilation time for the query once compiled with `nvcc` and once with NVRTC for five runs. As we can see, compiling the query with the runtime compiler is eight times faster. Thus, beyond the overhead of spawning another thread for compilation and the possibility of dynamic compilation, the runtime compiler decreases the compile time.

[3]https://nosferalatu.com/SimpleGPUHashTable.html

[4]https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/

TABLE I
COMPILATION TIME WITH NVCC AND NVRTC FOR QUERY IN LISTING 2.

	min [ms]	max [ms]
NVRTC	98	103
nvcc	796	823

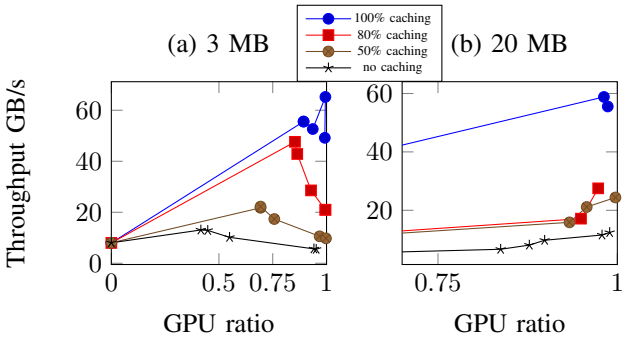

Fig. 4. Throughput for query in Listing 2 depending on the amount of data processed on GPU.

B. Runtime

Figure 4 visualises the throughput of the query from Listing 1 depending on the amount of data processed on the GPU (*GPU ratio*). The implementation follows work-stealing based on morsel-driven parallelism, so the GPU ratio can be steered by the number of available CPU threads and GPU streams.[5]

1) CPU-only: We start by investigating the performance on the CPU. When the dimension relations are each 20 MB in size, they exceed the size of our L3 cache. Therefore, almost every time we join a tuple from the sales relation with a tuple from the dimension relations, we get a cache miss. As a consequence, our CPU can only process 6.2 GB/s. When we reduce the size of the dimension relations to 3 MB each, substantial parts of them fit into the L3 cache on our CPU, which has 24.75 MB. Accordingly, the performance increases to 8.0 GB/s. This proves that caching plays a very significant role for the performance on the CPU.

2) GPU-only: Next, we investigate the performance on the GPU. Again, we start with dimension relations having a size of 20 MB each. When all data is replicated on the GPU, we observe a throughput of 22.9 GB/s which is almost 5x higher than on the CPU. When the sales relation is not replicated on the GPU (no caching), we observe a performance of 5.5 GB/s. When we reduce the size of the dimension relations to 3 MB each, the throughput increases to 65.2 GB/s (cached) and 5.5 GB/s, respectively (not cached). This shows that good spatial locality of reference improves performance on the GPU, albeit not as much as on the CPU. Still, having good performance even when the workload exhibits a bad locality of reference is a valuable benefit of GPUs compared to CPUs.

3) CPU/GPU: Finally, we examine the performance when parallelising across our CPU and our GPU. When no data of the sales relation is cached within the GPU memory, we observe a throughput of 13.0 GB/s. With caching, we achieve a throughput of 29.1 GB/s with 80 % processed on the GPU. This is easily explained as the sum of the throughput on the CPU (6.2 GB/s) and the throughput on the GPU (22.9 GB/s).

If we reduce the size of the dimension relations to 3 MB each, we can process 13.0 GB/s when no data of the sales relation is cached on the GPU. This already outperforms both the CPU-only (8.0 GB/s) and GPU-only (5.5 GB/s) versions. Caching data on the GPU increases the throughput. For instance, if 20 % of the data is cached, bandwidth increases to 15.7 GB/s. If we cache 100 % of the data, we get a performance of 65.2 GB/s.

[5] gitlab.rz.uni-bamberg.de/dt/gpudb-merzljak

V. CONCLUSION

This paper elaborated on the suitability of NVRTC for code-generating database system. We discussed the embedding of NVRTC within code-generation and its interaction with a tuple-at-a-time execution model. The evaluation showed that the compilation time for code-generation to GPU is accelerated by a factor of eight using NVRTC. Further, our hybrid approach based on work-stealing for morsel-driven parallelism scheduled the workload dynamically to CPU and GPU to achieve the highest possible throughput.

To summarise, NVRTC can be used for faster code-generation for GPU database systems in the future. Further, we argue that NVRTC will allow for dynamic compilation and loading during runtime after compilation. This can be used to compile code for GPU database systems at runtime as needed for adaptive query execution.

REFERENCES

[1] G. Graefe, "Volcano - an extensible and parallel query evaluation system," *IEEE TKDE*, 1994.
[2] T. Neumann, "Efficiently compiling efficient query plans for modern hardware," *VLDB*, 2011.
[3] A. Kemper and T. Neumann, "Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots," in *ICDE*, 2011.
[4] M. E. Schüle *et al.*, "Monopedia: Staying single is good enough - the hyper way for web scale applications," *VLDB*, 2017.
[5] V. S. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A low-level virtual instruction set architecture," in *MICRO*, 2003.
[6] M. A. Jibril *et al.*, "JIT happens: Transactional graph processing in persistent memory meets just-in-time compilation," in *EDBT*, 2021.
[7] M. E. Schüle *et al.*, "Freedom for the sql-lambda: Just-in-time-compiling user-injected functions in postgresql," in *SSDBM*. ACM, 2020.
[8] H. Rauhe, J. Dees, K. Sattler, and F. Faerber, "Multi-level parallel query execution framework for CPU and GPU," in *ADBIS*, 2013.
[9] J. Fett, A. Ungethüm, D. Habich, and W. Lehner, "The case for simdified analytical query processing on gpus," in *DaMoN*. ACM, 2021.
[10] E. A. Sitaridi and K. A. Ross, "Gpu-accelerated string matching for database applications," *VLDB J.*, 2016.
[11] ——, "Optimizing select conditions on gpus," in *DaMoN*. ACM, 2013.
[12] *NVRTC - CUDA Runtime Compilation*, 7th ed., NVIDIA, 5 Dec. 2014, https://www.wrfranklin.org/wiki/ParallelComputingSpring2015/cuda/nvidia/doc/pdf/NVRTC_User_Guide.pdf.
[13] S. Breß *et al.*, "Generating custom code for efficient query execution on heterogeneous processors," *VLDB J.*, 2018.
[14] J. Paul, B. He, S. Lu, and C. T. Lau, "Improving execution efficiency of just-in-time compilation based query processing on gpus," *VLDB*, 2020.
[15] M. E. Schüle *et al.*, "In-database machine learning with SQL on gpus," in *SSDBM*. ACM, 2021.
[16] ——, "Recursive SQL and gpu-support for in-database machine learning," *Distributed Parallel Databases*, 2022.
[17] P. Chrysogelos *et al.*, "Hetexchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines," *VLDB*, 2019.