Open, cross-architecture acceleration of data analytics with SYCL and RISC-V

Ivan Donchev Kabadzhov[1] José Morgado[2]
Aleksandar Ilic[2] Raja Appuswamy[1]

[1]Data Science Department, Eurecom, Biot, France
[2]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal

Abstract. The past few years have witnessed the growth in popularity of two standards for accelerating AI and analytics. On the hardware front, the advent of RISC-V, an open instruction set architecture, has ushered in a new era in standards-based design and customization of microprocessors. On the programming front, SYCL has emerged as a cross-vendor, cross-architecture, data parallel programming model for all types of accelerators. In this work, we take the first steps towards bringing these two standards together to enable a new line of work on fully-open, vendor-neutral, cross-architecture-accelerated database engines by developing SYCLDB–a SYCL-based library of key relational operations that works together with the oneAPI Construction Kit (OCK) to target multi-vendor CPU and accelerator backends. Using SYCLDB, we perform a comparative evaluation with micro and macrobenchmarks to show that SYCLDB can (i) exploit vectorization provided by RVV accelerators, (ii) provide performance on-par with CUDA counterparts on NVIDIA GPUs, and (iii) exploit multithreading in x64 and RISC-V CPUs, all the while using a single code base.

Keywords: SYCL · RISC-V · database · analytics · hardware acceleration

1 Introduction

The growing popularity of AI and analytics in virtually all application domains has led to a rapid increase in the amount of data gathered by enterprises and scientific institutions alike. Modern analytics and AI workloads are incredibly diverse and consist of a range of scalar, vector, and matrix computations. Thus, the traditional workhorse of the computing industry, the general-purpose CPU, cannot be optimized to meet the diverse requirements of such heterogeneous workloads [6]. This shortcoming of CPUs, in combination with the cessation of Dennard scaling, has led to a Cambrian explosion in the adoption of hardware acceleration solutions that can meet the demands of extreme scale AI and analytics workloads. Unfortunately, most popular solutions today, be it vertically-integrated, application-specific accelerators like Google TPU, or generalized, massively data-parallel SIMT processors like NVIDIA GPU, use proprietary, closed hardware—software stacks, leading to a monopolization of the analytics acceleration market by a few large industry players. Thus, there has been a growing interest in developing open, standards-based, cross-vendor approach to hardware acceleration as witnessed by the rise in popularity of two key standards, namely, RISC-V and SYCL.

Over the past few years, RISC-V, a standardized, free, open Instruction Set Architecture (ISA), has emerged as an alternate solution in the AI acceleration space. At its core, RISC-V has a very simple ISA with 47 instructions. But application verticals can customize the ISA with optional extensions that are often realized through specific licensing deals between a RISC-V vendor and the licensee. Thus, RISC-V creates a new business model that allows companies to provide commercial support and other deliverables that are needed for the processor verification or integration into a chip. Further, the RISC-V ISA, with recent extensions like support for IEEE-754 floating-point standard, and the highly customizable RISC-V vector extension (RVV), has emerged as a viable alternative to x64 and ARM for supporting compute- and data-intensive workloads. For instance, SG2042 is a RISC-V CPU designed for high performance workloads [3].

As the RISC-V ecosystem continues to evolve, we are also starting to see RISC-V accelerator solutions. This naturally necessitates an open programming model that supports a cross-vendor, cross-architecture computation offloading. Over the past few years, SYCL has emerged as a higher-level programming model that brings hardware acceleration to mainstream C++ with the goal of making it easy to write new software similar to CUDA. Expressing data-parallel computations in SYCL provides two key advantages over other competing programming models. First, SYCL builds on a portable high-level programming language by implementing an open standard in contrast to other data-parallel languages that are proprietary (CUDA) or low level in nature (OpenCL). In doing so, SYCL enables the development of portable, cross-architecture applications. Second, SYCL is a single-source by nature as it allows host and device code to co-exist in a single file unlike other programming models. This allows the SYCL compiler the ability to analyze and optimize across the entire program regardless of the device on which the code is to be run.

In this work, we take the first steps towards bringing together these two standards to enable the development of a new generation of data analytics engines that can support and exploit open, standards-based hardware acceleration. We present SYCLDB, a library of relational primitives implemented in SYCL, that can be used to execute SQL operations on RISC-V CPUs and RVV accelerators. Using SYCLDB, we perform an experimental evaluation of a real RISC-V CPU (SG2042) and present the first Cache-aware Roofline Model (CARM) analysis of a database workload on a RISC-V CPU to understand (i) how effectively SYCLDB exploits underlying hardware, and (ii) what are the bottlenecks in executing SYCLDB queries. We also provide an evaluation and analysis of an x64 CPU as a point of comparison. On the accelerator side, we demonstrate the ability to execute SYCLDB on an FPGA-based RISC-V accelerator platform. In addition, we also present an evaluation of using SYCLDB on NVIDIA GPUs to show that it can match the performance of its state-of-the-art CUDA counterparts. Finally, we outline new opportunities provided by SYCL and RISC-V in developing performance-portable, cross-architecture-accelerated database engines. We make SYCLDB open source [1] to enable further research on these topics.

[1] https://github.com/SYCLOPS-Project-EU/SYCLDB

2 Design

In this section, we first present the design of SYCLDB while highlighting relevant aspects of SYCL. Then, we describe the infrastructure based on oneAPI Construction Kit used to execute SYCLDB queries on RISC-V CPUs and RVV accelerators.

2.1 SYCLDB Relational Operations

Primitive functions.

SYCLDB is structured around a set of low-level primitives designed to implement relational operators efficiently across heterogeneous hardware. The current set includes selection, hashtable build, hashtable probe, and aggregation. These primitives follow established design patterns from prior database acceleration efforts [13] and can be composed to support complex operations.

Selection supports multiple conditions by scanning each tuple only once. Instead of materializing intermediate results per condition, we use a compact flag array to track which tuples satisfy the conjunction of conditions.

Join is implemented using a combination of **build** and **probe** primitives. The build primitive constructs a hash table using the smaller relation, employing linear probing for its simplicity and regular memory access patterns. The probe primitive scans the larger relation and performs lookups in parallel using a remainder-based hash function.

Aggregation relies on the `reducer` abstraction introduced in the SYCL 2020 revision, enabling efficient computation of sum, product, minimum, and maximum over trivially copyable C++ types.

Implementing SQL queries. SYCLDB is not a full-fledged database engine. Constructing a cross-architecture-accelerated database engine with SYCLDB requires several other components, like a SQL query parser to produce a query plan, an optimizer to perform cost-based optimization of the plan, etcetera. In this work, we focus on studying the individual SYCL-based relational primitives in SYCLDB rather than focusing on a full-fledged database implementation. Thus, similar to other CUDA-based acceleration libraries [13], SYCLDB implements SQL queries by directly invoking relevant primitives instead of parsing and interpreting SQL.

Real-world queries often involve complex multi-way joins and conjunctive selection predicates. A naive implementation might assign each primitive to a separate kernel, submitted independently to the SYCL queue. However, this strategy incurs high overhead from repeated kernel launches and necessitates full materialization of intermediate results. Instead, SYCLDB adopts a pipelined execution strategy, combining multiple primitives within a single `parallel_for`. For instance, a selection primitive operates per-tuple, evaluating predicates and updating a local boolean flag. If the tuple satisfies all conditions, it is immediately passed to the aggregation logic within the same kernel. This approach avoids redundant memory accesses and eliminates intermediate materialization by maintaining selection state in-kernel. Listing 1 illustrates this pattern.

Minimizing Transfer Overheads and Scaling to Multiple Accelerators Prior work on GPU-accelerated database design has demonstrated that in order to achieve maximum performance with GPUs, it is necessary for all data to be GPU resident [13].

```
template <typename T> void select_primitive( // acting on 1 element
  bool &sf, T c, logic_op log /*AND, OR*/, comp_op comp /*<, >*/, T v){
  logic(log, sf, compare<T>(comp, c, v));
}
q.parallel_for(F.len, [=](sycl::id<1> i...){
    bool sf; // selection flag (accumulates condition)
    select_primitive<float>(sf, f1[i], NONE, GT, 3);  // sf=(f1[i]>3)
    select_primitive<float>(sf, f2[i], AND, LT, 5); // sf=sf&&(f2[i]<5)
    // if row satisfies predicate, add f3[i] to sum
    if (sf) sum.combine(f3[i]);
}
```

Listing 1: SYCLDB query corresponding to the SQL statement SELECT SUM(f_3) FROM F WHERE $f_1 > 3$ AND $f_2 < 5$

However, as each GPU has limited onboard memory, it might not be feasible in all cases for data to reside on a single GPU. Thus, some GPU database engines use host CPU memory to store all data and perform staged query execution. SYCLDB also supports such staged execution by partitioning data stored in CPU memory into chunks such that it can overlap the transfer of one chunk with the kernel execution of another chunk. For some operators, like selection or projection, this is relatively straightforward to implement; columns that are input to these operators are partitioned on the fly to enable overlap. However, a join operation cannot be arbitrarily partitioned as it has distinct build and probe phases. Typical analytical databases often follow a star schema [11] that result in join queries having one or more dimension tables that are very small, and one large fact table. Distributed database engines [5] perform query execution across nodes by first creating a copy of the small dimension tables on each node, using them to build a hashtable in parallel. The probe table is then partitioned across nodes to perform the join in parallel. SYCLDB follows a similar strategy by building a hash table on each accelerator. The probe table, in contrast, is partitioned into distinct chunks, with each accelerator processing a unique subset of chunks. In terms of implementation, we use SYCL *events* to overlap data transfers and kernel execution.

2.2 SYCLDB on RISC-V with OCK

As mentioned earlier, one of the goals of our work is to investigate the suitability of SYCL for exploiting RISC-V CPUs and more importantly, offloading database operations to RVV accelerators. We use the recently open-sourced oneAPI Construction Kit (OCK) to enable this. OCK provides the OpenCL driver for RISC-V that enables running SYCL kernels on the RISC-V CPU. We use it to execute SYCLDB queries on the 64-core SG2042 RISC-V CPU. For offloading SYCLDB queries to custom accelerators (including RVV accelerators), OCK provides supporting infrastructure to enable the development of an accelerator-specific OpenCL backend using a Hardware Abstraction Layer (HAL). In reality, an RVV accelerator would be an ASIC. However, for this work, we relied on SYCLARA [1]–an end-to-end, hardware–software testbed consisting of a CVA6 RISC-V soft core [14] and ARA2 [4] RVV1.0 implementation running on a Xilinx Virtex UltraScale+ FPGA. SYCLARA also contains relevant support to

enable booting and running Linux with networking support on the RISC-V soft core. To offload SYCLDB queries to SYCLARA, we relied on the OCK remote Hardware Abstraction Layer which provides a server and a client, which are programs that can be run on any standard Linux installation, and use socket connections to communicate with each other. We run the OCK server on the FPGA Linux distribution and the OCK client on an x64 machine. The OCK client provides an OpenCL backend that when targeted, will result in SYCL kernels being JIT compiled to generate RVV instructions that are forwarded for execution to the server. Using this approach, when SYCLDB is executed targeting the OCK client, its queries get offloaded to the OCK server, and the generated RVV instructions are executed on SYCLARA RVV accelerator.

3 Evaluation

In this section, we will present the results of our experimental evaluation. Our goal is to answer the following questions: (i) how does SYCLDB scale on NVIDIA GPU and compare with CUDA-based, state-of-the-art Crystal library [13]?, (ii) how well does SYCLDB utilize RISC-V CPUs? and (iii) how well does SYCLDB benefit from offloading its query to the RVV accelerator?

3.1 Setup

Hardware and Software Setup. Our hardware testbed consists of the following:

- A x86-64 server equipped with a 24-core Intel(R) Core(TM) i9-10920X Cascade Lake CPU clocked at 3.50GHz and 128GB of DDR4 RAM. The system runs Ubuntu 22.04.5 LTS. We compile SYCLDB with ACPP compiler (v25.02.0).
- Two NVIDIA GeForce L40S GPUs accessed via a PCI-Express Gen4 x16 interface, which provides a theoretical peak bandwidth of 31.5 GB/s in each direction. We use the same ACPP v25.02.0, built with NVCC v12.4 support, to target the GPUs.
- A System-on-Chip equipped with a 64-core SG2042 RISC-V CPU clocked at 2GHz and 128 GB of DDR4 RAM. We use DPC++ from `cross_riscv` branch of the `PietroGhg/llvm` repository [2], specifically the commit f56b10a.
- The SYCLARA RISC-V accelerator. We use the DPC++ compiler (v2024-06-03) to cross compile SYCLDB, and communicate with the device via OCK v4.0.0.

Benchmarks. For the workload, we generate a synthetic database containing two tables F and D. Table F is a large fact table containing two columns f_1 and f_2 both of which are 4-byte floating point values. Table D is a small dimension table containing two columns d_1 and d_2 which are also 4-byte floating point values. We consider two queries: (i) a projection query, which computes a linear combination of the columns on the fact table, i.e. SELECT $a \times f_1 + b \times f_2$ FROM F, and (ii) a join query that performs SELECT SUM $(f_1 \times d_1)$ FROM F, D WHERE $f_2 = d_2$. For all experiments, the D table is fixed at 128MB and the F table is 2GB, following the setup used in prior Crystal

[2] https://github.com/PietroGhg/llvm/tree/pietro/cross_riscv

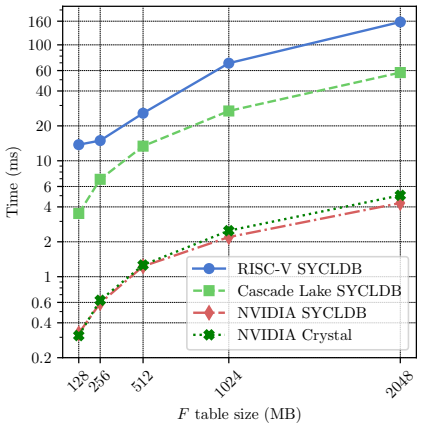

Fig. 1. Projection query execution time

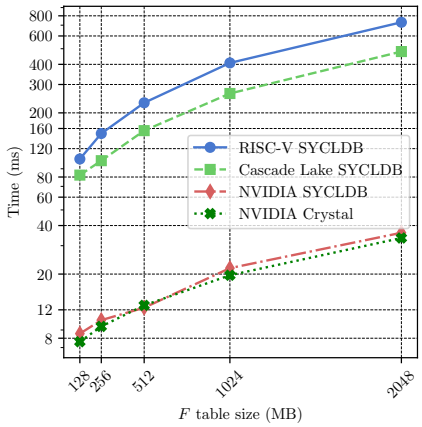

Fig. 2. Join query execution time

evaluations [13]. Each experiment is run five times. We discard the first cold run and report the average of the remaining four. Error margins are omitted as runtime variation is consistently under 5%.

3.2 Cross-architecture performance of SYCLDB on CPU and GPU backends

Figures 1 and 2 show the execution time of SYCLDB and Crystal on various CPU and GPU backends. While SYCLDB can support out-of-GPU data access, Crystal is designed explicitly for scenarios where all data sits on the GPU. As our goal here is to evaluate SYCLDB and Crystal side by side, all data was made GPU resident for both systems. There are several important observations to be made. First, comparing SYCLDB with Crystal, we see that their performance is identical under both queries. Prior work comparing SYCL versus CUDA for database acceleration had shown CUDA-based implementation to be 4× faster than a SYCL-based one [9]. Our result shows that this gap has been closed, thanks to improvements in SYCL compiler toolchains. Second, comparing CPU and GPU, we can see that both queries exhibit a similar trend; the NVIDIA GPU is 11× faster than the x64 CPU, which in turn, is 2.5× faster than the RISC-V CPU for the projection query, and 1.4× faster for the join query. This can be explained by the fact that the memory on the NVIDIA GPU has an order of magnitude higher memory bandwidth compared to the x64 and RISC-V CPUs. Similarly, SG2042 is one of the earliest RISC-V CPU tape outs available in the market. Given this, it is understandable that its performance is not competitive with the more mature x64 CPU.

Next, we present an evaluation of SYCLDB when data is not GPU resident. Figures 3 and 4 show the result of executing the two queries on one/two GPU(s) in two scenarios: (i) when data is entirely in GPU memory, and (ii) when the data sits in host CPU memory. For the former case, the fact table F is partitioned across the two GPUs, and the dimension table D is copied in each GPU. Thus, SYCLDB executes both projection and join queries in parallel on both GPUs. For the latter case, when data is in CPU memory, SYCLDB partitions the data on the fly and performs staged query execution to

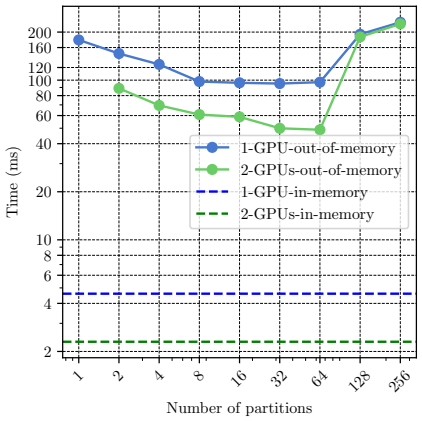

Fig. 3. Projection query execution time on GPU in various data residency scenarios

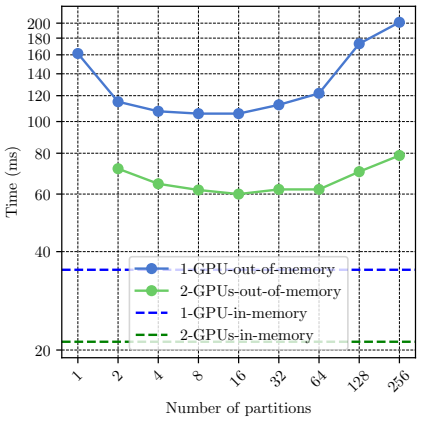

Fig. 4. Join query execution time on GPU in various data residency scenarios

overlap PCIe data transfers with kernel execution. Here, we present the execution time of both queries while varying the number of partitions.

First, focusing on the cases where data is in GPU memory, in the projection query, we can observe a 2× reduction in execution time scaling from one to two GPUs. Whereas, for the join query, each GPU would need to pay the constant cost of 6ms for the build table creation. Isolating the build time, we reach 2× improvement. This shows that SYCLDB can effectively use multiple accelerators for parallel query execution. Second, when data is in host memory, we can see that the performance of staged query execution improves with more partitions due to better overlap between data transfers and kernel execution. However, using excessively large numbers of partitions introduces two bottlenecks: (i) decreased PCIe bandwidth efficiency due to smaller transfer sizes, and (ii) increased kernel launch overhead. Currently, we determine the optimal number of partitions manually, balancing transfer and execution overlap against these overheads. In future work, we aim to automate this based on runtime profiling and hardware characteristics.

Finally, the rate of improvement achieved via overlap in staged execution differs between projection and join queries. For the projection query, execution time drops from 178.5ms in the non-partitioned case to 95.2ms with partitioning, resulting in a 1.88× speedup. The observed PCIe host-to-device and device-to-host bandwidths on our hardware was in the range of 22-23GB/s. For this query, the input consists of two columns amounting to 2GB. Thus, just the transfer of the 2GB input table on which the projection is being done will take around 90ms. The output of the projection query consists of one column amounting to 1GB. Thus, the device to host transfer would take an additional 45ms. Hence, even excluding the kernel execution time, the projection query without any overlapped transfers would take 135ms. In contrast, our execution of 95.2ms shows that SYCLDB is able to overlap the host-to-device transfer with device-to-host transfers and kernel execution to reduce overall execution time. For the join query, the total runtime improves from 161ms to 105.75ms, that is a 1.52× speedup. The speedup is less than what was achieved with projection due to the fixed overhead of

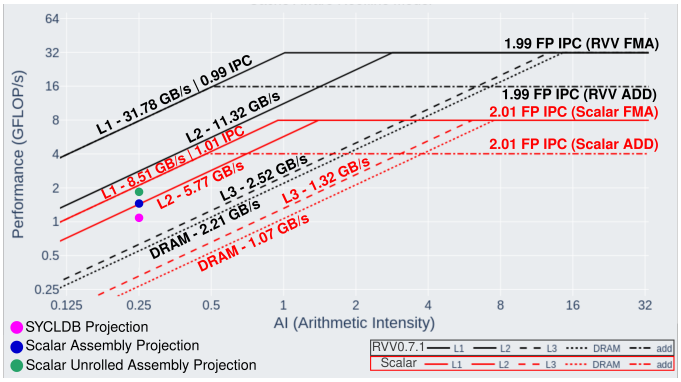

Fig. 5. CARM on RISC-V CPU

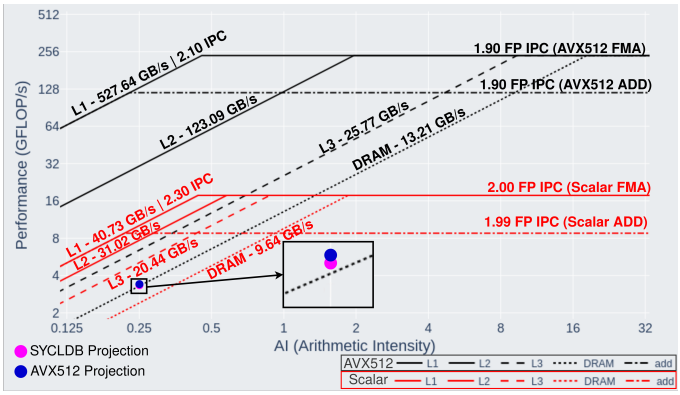

Fig. 6. CARM on x64 CPU

12 ms for build-side transfers and 6 ms for building the hash table. In addition, the probe phase incurs a lot of random memory accesses due to hash table lookups compared to the sequential scans performed with projection.

3.3 CARM analysis on CPUs

Having presented a comparative evaluation of SYCLDB on various processor backends, we now answer the question "how well does SYCLDB utilize the RISC-V CPU?" by presenting a Cache-Aware Roofline Model (CARM) analysis using the CARM Tool [7, 10]. From a theoretical standpoint, the performance of any SYCLDB query can be limited by two architectural upper-bounds: *i)* the peak compute performance, typically for Floating Point (FP) operations in GFLOPS; and *ii)* peak memory bandwidth of various memory levels. Regarding the maximum compute performance, both tested RISC-V and Intel CPUs are capable of retiring up to two FP instructions per cycle (IPC), due to the presence of 2 FP units per core, both capable of performing FP SIMD instructions from the widest supported extension (RVV or AVX512). On the memory side, the Intel CPU contains three load/store units capable of delivering up to three memory IPC, while the SG2042 only contains one load/store unit per core offering a maximum of

one memory IPC. However, these theoretical limits might not always be achievable in their respective systems, thus we rely on the tailored microbenchmarks integrated in the CARM Tool to accurately evaluate these limits [10]. In this section, we first present a CARM analysis of RISC-V and x64 CPUs to understand these limits. Then, we present a CARM analysis of SYCLDB queries to identify how well they perform relative to these limits.

Figure 5 shows the results obtained from the CARM tool under the SG2042 RISC-V CPU. The figure shows the bandwidths and peak GFLOPS obtained for both Scalar (red lines) and RVV (black lines) using single precision FP operands on 1 thread. As can be seen, the use of RVV results in approximately 4 times higher L1 bandwidth and peak GFLOPS for both the Add and Fused-Multiply-Add (FMA) instructions, which is expected since this RISC-V CPU implements RVV operands with a width of 16 bytes, capable of executing instructions with up to four single precision FP operands. We can also notice that for the higher memory levels the performance gain from RVV lowers to only double of the Scalar performance. We can also verify from the IPC reported for both L1 and peak GFLOPS that the theoretical limits outlined previously were achieved with an average deviation of around 0.5%.

The CARM plot for the x64 Intel Cascade Lake CPU is presented in Figure 6, containing both AVX512 (black lines) and Scalar (red lines) roofs. We can observe substantial performance improvements from AVX512, with gains of 16 times for peak GFLOPS and 13 times for L1 bandwidth when compared to Scalar, with diminishing benefits in higher memory levels. We can also observe that 2.10 and 2.30 IPC for L1 cache were achieved for AVX512 and Scalar, respectively. Although these values do not reach the theoretical value of three memory IPC, they are in-line with the "sustained bandwidth" reported by Intel for the Skylake-X architecture (2.08 IPC) [8].

Following the CARM-based hardware evaluation, we now explore the performance of the SYCLDB projection query on both CPUs in relation to the architectural performance limits. The projection query consists of three FP operations (two multiplications and one addition) that require two loads and one store operation for every element, which results in an Arithmetic Intensity of 0.25. By measuring the GFLOPS performed during the execution of the projection query, CARM tool can plot its performance on the CARM graph for each system, as presented in Figures 5 and 6. As mentioned earlier, the RISC-V CPU supports both scalar instructions and SIMD computation with vector instructions. However, the vector instructions supported are based on RVV v0.7.1 specification, which is deprecated with the official ratified standard being RVV 1.0. Thus, no SYCL compiler today has complete support for RVV0.7.1. As a result, it was not possible to compile SYCLDB to generate vectorized RVV instructions and obtain what could be the peak vectorized performance of the projection query on the RISC-V CPU. Instead, we were only able to generate scalar RISC-V instructions. Thus, we focus our analysis on the scalar implementation of the projection in RISC-V. The FPGA accelerator results presented in Section 3.4 cover SYCLDB's ability to exploit RVV 1.0 to improve performance.

To gain further insight on the performance difference between SYCL and assembly-specific implementations, we also developed two customized, RISC-V-assembly-based functions. The first assembly function (referred to as Scalar Assembly in Figure 5) uses

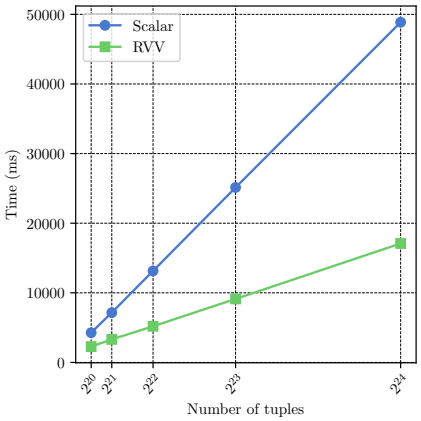

Fig. 7. Projection times on a RISC-V accelerator

scalar RISC-V instructions to implement the projection query directly in assembly. The second (referred to as Unrolled Assembly) applies manual unrolling to develop a more optimized version that can provide performance closer to the peak scalar performance. Figure 5 shows that the scalar assembly function achieves a 1.3× speedup compared to the SYCL implementation. We can also see that the unrolled scalar approach improves performance even more, and is very close to architectural limits defined by the CARM for scalar instructions. We believe that the difference in performance between SYCL and assembly can be explained by the fact that SYCL compilation on RISC-V follows the intermediate SPIR-V path, with the SPIR-V code being JIT compiled by the OCK OpenCL runtime. It is well known that this path adds some overhead. Recently, a Native CPU compilation path has been introduced in DPC++ compiler for x64 CPUs where the kernel code is not compiled to an intermediate SPIR-V, but rather directly to x64 assembly and linked with the rest of the program as a shared library. We will show the CARM analysis of x64 CPU next using this Native CPU compilation path, but unfortunately, this path is not available for RISC-V host CPUs.

Figure 6 presents the CARM analysis on the Intel CPU. Similar to SG2042, we also developed a customized, hand-coded AVX512 assembly to see the difference with SYCL code. Here, the hand-coded implementation resulted only in marginal speed-up when compared with the SYCL implementation. This indicates that the different SYCL compiler target architectures (x64 versus RISC-V) can significantly impact the performance optimization of the same SYCL code, leading to a better exploration of the x64 CPU when compared with the RISC-V CPU, and that SYCLDB queries are able to effectively exploit vectorization when targeting x64 CPUs but not RVV0.7.1 on RISC-V. Thus, this clearly highlights the need for further work on compiler support for RVV, especially given the newly ratified RVV 1.0 standard.

3.4 SYCLDB on RISC-V accelerator

Finally, Figure 7 analyzes the scalability of SYCLDB's projection operator on a RISC-V accelerator. As discussed in Section 2.2, this evaluation uses the SYCLARA platform,

which integrates the CVA6 RISC-V soft core with the ARA2 implementation of RVV 1.0. While the SYCLDB benchmark is launched from an x64 host, kernel execution is remotely offloaded to SYCLARA using the OCK HAL interface. Due to hardware constraints—specifically, a 1GB memory limit and the overhead of a full Linux OS on board – we restrict input size to a fact table of size 128MB (2^{24} tuples).

We evaluate projection in two configurations: (i) using RVV-enabled vector execution on ARA2, and (ii) using scalar execution on CVA6 without RVV. As Figure 7 shows, RVV achieves up to a 2.8× speedup over scalar execution, highlighting that the SYCL compiler toolchain is capable of generating efficient vector code for RVV targets, while execution time scales linearly with input size in both modes.

Compared to x64 CPUs and GPUs (Figures 1 and 2), the absolute performance of the RISC-V FPGA is one to two orders of magnitude lower, which is expected due to the use of a soft core, limited memory bandwidth, and remote kernel offloading. Nevertheless, this experiment serves to demonstrate the portability of SYCLDB to RISC-V vector accelerators via OCK, rather than to compete on raw throughput.

4 Future Work & Conclusion

The growing popularity of RISC-V and SYCL has made it feasible to create an open, standards-based stack for data analytics and AI. In this work, we took the first step in bringing these two standards together by developing SYCLDB. Our work opens up several avenues of future research. On the RISC-V front, this is the investigation of RVV 1.0, the ability of SYCLDB to exploit it, and its comparison to SIMD support in other architectures. More broadly, the ability to modify the ISA with RISC-V opens up several opportunities for hardware–software codesign, from developing custom ISA extensions for functionalities like compression/decompression, to developing RISC-V-based near-data processors and computational storage solutions. On the SYCL front, further work is required to understand if newly introduced functionalities, like kernel fusion that enables multiple SYCL kernels to be fused into one at runtime [12], or SYCL graphs that enable computations across several kernels to be scheduled together, can be used to improve query execution on CPUs and GPUs [2]. Recently, SYCL implementations like AdaptiveCPP have started to add support for automatic, cross-device kernel execution. Thus, another line of work is investigating policy–mechanism split that can enable database engines to share information with SYCL runtimes, and vice versa, to enable automatic query execution on multi-vendor accelerators.

Acknowledgments. This work was supported by the Horizon Europe Project “SYCLOPS”, funded by the European Union HE Research and Innovation programme under grant agreement No. 10109287. Additional support was provided by FCT (Fundação para a Ciência e a Tecnologia, Portugal) through UIDB/50021/2020 project.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Bilandi, M.R., Kabadzhov, I.D., Appuswamy, R.: Syclara: An open hardware-software platform for evaluating sycl applications on risc-v vector accelerators. IWOCL 2025, 13th International Workshop on OpenCL and SYCL, 7-11 April 2025, Heidelberg, Germany (2025), © ACM, 2025. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in IWOCL 2025, 13th International Workshop on OpenCL and SYCL, 7-11 April 2025, Heidelberg, Germany https://doi.org/10.1145/3731125.3731136
2. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: GPU-Accelerated Database Systems: Survey and Open Challenges, vol. 8920, pp. 1–35. Springer (12 2014)
3. Brown, N., Jamieson, M., Lee, J., Wang, P.: Is risc-v ready for hpc prime-time: Evaluating the 64-core sophon sg2042 risc-v cpu. In: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. p. 1566–1574. SC-W '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3624062.3624234, https://doi.org/10.1145/3624062.3624234
4. Cavalcante, M., Schuiki, F., Zaruba, F., Schaffner, M., Benini, L.: Ara: A 1-ghz+ scalable and energy-efficient risc-v vector processor with multiprecision floating-point support in 22-nm fd-soi. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **28**(2), 530–543 (2020). https://doi.org/10.1109/TVLSI.2019.2950087
5. Chen, J., Jindel, S., Walzer, R., Sen, R., Jimsheleishvilli, N., Andrews, M.: The memsql query optimizer: a modern optimizer for real-time analytics in a distributed database. Proc. VLDB Endow. **9**(13), 1401–1412 (Sep 2016). https://doi.org/10.14778/3007263.3007277, https://doi.org/10.14778/3007263.3007277
6. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., Burger, D.: Dark silicon and the end of multicore scaling. SIGARCH Comput. Archit. News **39**(3), 365–376 (Jun 2011). https://doi.org/10.1145/2024723.2000108, https://doi.org/10.1145/2024723.2000108
7. Ilic, A., Pratas, F., Sousa, L.: Cache-aware roofline model: Upgrading the loft. IEEE Comput. Archit. Lett. **13**(1), 21–24 (Jan 2014). https://doi.org/10.1109/L-CA.2013.6, https://doi.org/10.1109/L-CA.2013.6
8. Intel: Intel 64 and ia-32 architectures optimization reference manual volume 1. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html (2024)
9. Marinelli, E., Appuswamy, R.: Xjoin: Portable, parallel hash join across diverse xpu architectures with oneapi. In: DAMON (2021)
10. Morgado, J., Sousa, L., Ilic, A.: Carm tool: Cache-aware roofline model automatic benchmarking and application analysis. In: Proceedings of the IEEE International Symposium on Workload Characterization. p. 1–11. IISWC'24, IEEE, New York, NY, USA (2024), https://github.com/champ-hub/carm-roofline
11. O'Neil, P., O'Neil, E., Chen, X., Revilak, S.: The Star Schema Benchmark and Augmented Fact Table Indexing, p. 237–252. Springer-Verlag, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-10424-4_17
12. Pérez, V., Sommer, L., Lomüller, V., Narasimhan, K., Goli, M.: User-driven online kernel fusion for sycl. ACM Trans. Archit. Code Optim. **20**(2) (mar 2023). https://doi.org/10.1145/3571284, https://doi.org/10.1145/3571284
13. Shanbhag, A., Madden, S., Yu, X.: A study of the fundamental performance characteristics of gpus and cpus for database analytics. In: SIGMOD (2020)
14. Zaruba, F., Benini, L.: The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **27**(11), 2629–2640 (Nov 2019). https://doi.org/10.1109/TVLSI.2019.2926114