Cross-architecture High-performance Acceleration of Data Analytics Operators with SYCL and OneAPI DPC++

Massimiliano Pronesti
Institut EURECOM
Biot, France
massimiliano.pronesti@gmail.com

Giulio Corallo
Institut EURECOM
Biot, France
corallo.giulio@yahoo.it

Raja Appuswamy
Institut EURECOM
Biot, France
raja.appuswamy@eurecom.fr

ABSTRACT

Modern server hardware is increasingly heterogeneous with a diverse mix of XPU architectures deployed across CPU, GPU, and FPGAs. However, heterogeneous hardware requires tedious optimization of DBMS algorithms for each platform it supports when implemented with vendor-specific toolchains like CUDA or OpenCL. Moreover, dealing with FPGAs and ASICs often requires an higher knowledge of hardware design and hardware description languages (HDL) which often makes the developement board-specific.

Such an approach inevitably leads to specialization and maintainability issues and the lack of portable parallelism caused by the absence of a common high-level programming framework is the main reason preventing a wider adoption of XPUs for database systems.

In this paper, we take the first steps towards solving this problem using oneAPI –a cross-industry effort for developing an open, standards-based unified programming model that extends standard C++ to provide portable parallelism across diverse processor architectures. In particular, starting from the Crystal library [12], we aim at extending the work done in [8] for the hash join to other data analytics operators and queries and to trace their performances on multicore CPUs, integrated GPUs (Intel GEN9), discrete GPUs (Intel DG1 and NVIDIA GeForce), as well as FPGAs (Intel Arria and Strattix). We compare the performance of operators using DPC++ and hipSYCL [4] to demonstrate the performance–portability trade offs. Finally we perform a workload evaluation at query level.

1 INTRODUCTION

The end of Dennard scaling and the rising popularity of data analytics and machine learning have resulted in a rapid increase in the adoption of heterogeneous parallelism. Graphics Processing Units (GPU) and Field Programmable Gate Arrays (FPGA) have evolved from being used as accelerators in niche application areas to being an integral part of almost all cloud computing platforms. This has led to a surge in interest in the design of database systems that can exploit such XPU architectures instead of the CPU.

Recent development of highly efficient GPU-specific algorithms for common relational operators have shown significant performance benefits over classical CPU-tailored implementations. Supporting the variety of hardware accelerators in a single query engine using vendor-specific tool-chains is a tedious effort that inevitably leads to specialization and prioritization of certain hardware types/models (for instance, CUDA [9]) and prevents efficient scaling-up. Using a high-level language that abstracts hardware features yet allows efficient hardware utilization would significantly simplify support and development. Recent work has investigated the performance–portability trade offs in using SYCL for accelerating key HPC applications [6]. In this paper we investigate the possibility of using a high-level abstraction for performance-portable database engines by focusing on three of the most important DBMS algorithms - hash-join, select, project - and implications on the design. We compare several associative container implementations in SYCL language across different devices and evaluate improvement potential using theoretical bounds estimations.

2 BACKGROUND

Graphic processing units (GPUs) have a high computational power.A GPU is designed with large amount of computing units running simultaneously, so it can perform parallel operations on multiple sets of data. Using these advantages, calculations on GPUs now also commonly used for nongraphical tasks.

SYCL [11] is Khronos standard adding data parallelism to C++ for heterogeneous systems. DPC++ is an open source compiler project based on SYCL, a few extensions, and broad heterogeneous support that includes GPU, CPU and FPGA support. SYCL execution model is based on Standard Portable Intermediate Representation (SPIR-V). This allows to compile the same code for different types of devices.

FPGAs, differently from GPUs, are commonly classified as a spatial architecture. Programs that are spatially implemented theoretically take the program as a whole and display it all at once on the device. In spatial architectures, each instruction receives its own dedicated hardware that can execute concurrently. Due to the device's tremendous parallelism, program execution on the device can be extremely efficient if a program employs the majority of the FPGA's storage and there is enough work to keep all of the hardware active every clock cycle. With an FPGA, the utilization of area may be properly suited to a single application without waste, in contrast to more general architectures that may have considerable amounts of wasted hardware per clock cycle. Through huge parallelism and typically impressive energy efficiency, this modification can speed up applications. The drawback: To fit on a device, large programs might need to be tuned and reorganized. Compiler resource sharing features can help with this, but typically at the cost of some performance reduction, which lessens the advantages of employing an FPGA. One of the powers of spatial architectures is pipelining [11], which means that execution of a single program is spread across many clock cycles. Like most modern compute accelerators, achieving good performance requires a large amount of work to be performed, consequently, even though we provide some insights into the implementations of various operators on FPGAs, the main contribution of this paper is to provide developers with a roadmap of the types of performance differences one should anticipate seeing in database operators implementations. Compiling your FPGA design can take a long time, from 5 to 12 hours.

To be productive therefore it is recommended, even by Intel[1], to follow the steps in Figure 1.

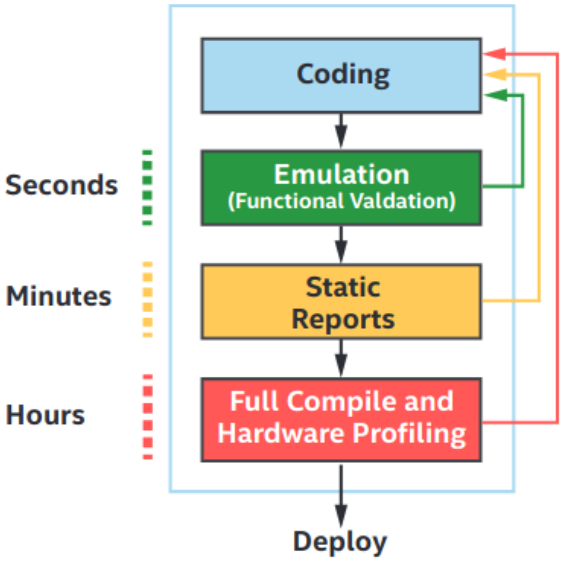

Figure 1: FPGA Development Flow

There is a perfect pipelining when each stage of the pipeline is occupied and doing useful work every clock cycle as described in Figure 2.

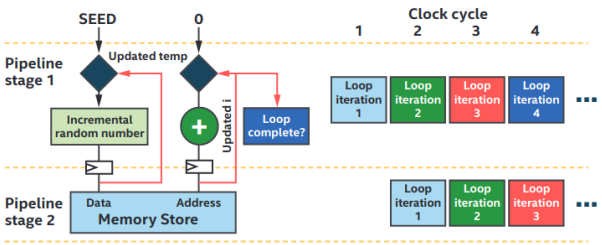

Figure 2: Pipeline

However, because a data dependence may require numerous clock cycles to compute in real algorithms, it is frequently not practical to start a new loop iteration every single clock cycle. When memory lookups, especially those from off-chip memories, are involved in the computation of a dependency, this problem frequently occurs. As a result, the pipeline has an initiation interval (II) of N clock cycles and can only start a new loop iteration once per N clock cycles. An II larger than one can lead to inefficiency in the pipeline because the average occupancy of each stage is reduced. Restructuring the compute in a loop based on the reports can often reduce the II. Another important concept in spacial architectures is Pipes. They allow us to decompose a problem into smaller pieces to focus on development and optimizations in a more modular way.

3 SYSTEM DESIGN

In this section, we first provide an overview of the Crystal tile-based execution approach. Then, we detail how we ported it from CUDA to DPC++. Eventually, we describe custom FPGA implementations leveraging that specific accelerator.

3.1 Crystal Tile-based Execution Model

The idea behind tiling comes from the observation that threads in a GPU are grouped into thread blocks (in CUDA terminology) such that threads within a thread block can communicate through the shared memory and synchronize through barriers. Hence, even though a single thread on the GPU at full occupancy can hold only up to 24 integers in shared memory, a single thread block can hold a significantly larger group of elements collectively between them in shared memory. The set of data elements that can be collectively processed by a thread block is referred to as a **tile**. In this fashion, it is possible to design kernels in terms of block-wide device functions dealing with a set of tiles as units of input and output. Each function uses vector instructions for memory accesses, and registers for storing values.

One key advantage of this approach is that after a tile is loaded into the shared memory, subsequent passes over the tile will be read directly from shared memory and not from global memory, avoiding the second pass through global memory described in the implementation above.

3.2 DPC++ Porting

In order to migrate the original CUDA implementation of Crystal to DPC++, we start with Intel's official migration tool, which allows to convert CUDA code to DPC++ at syntax level recognizing the main CUDA constructs and converting them to their DPC++ equivalent. Our goal in using the compatibility tool is to understand and document issues in converting various aspects like data movement, kernel parameterization, atomics and synchronization from CUDA into DPC++,

Using the Compatibility Tool involves the use of the command dpct that takes a .cu file as input and produces its DPC++ counterpart, with dp.ct extension. Thus, we apply the command to all .cu file of the project. At the source level, the overall translation is quite accurate. dpct automatically adds necessary boilerplate such as headers and compiler directives required for enabling DPC++ compilation. Similarly, dpct preserves and converts templatized functions that correspond to block primitives and join kernels of the Crystal library for most part, with some minor syntactic modifications. At the programming model level, dpct replaces CUDA kernel launches with an nd_range parallel_for kernel. Further, CUDA data management calls that move data from host to device memory, or assign specific values to device allocated memory regions, are replaced with appropriate DPC++ calls (memcpy and memset functions of the DPC++ queue class).

Despite its utility, dpct does not convert everything automatically and correctly and the additional porting work to be done by the developer is not negligible. The first issue concerns kernel dimensions. CUDA programming model requires kernel dimensions to be specified in terms of number of threads in a thread block, and the number of thread blocks per grid. Moreover, both thread

blocks and grids can be multidimensional. Similarly, DPC++ uses the notion of work-item and work-group. Thus, a CUDA thread block roughly corresponds to a DPC++ work-group, and a CUDA thread gets mapped to a work-item in DPC++. DPC++ also provides an nd_item object to enable index lookup in a nd_range kernel. It represents the index of each work-item. However, despite the fact that the original code implements a 1D kernel, dpct converts it into 3-dimensional kernel. As consequence, all accesses to the threads indexes (local-id, global-id, group-id) within the kernel code were wrong and needed to be rewritten. Second, synchronization primitives and low-level constructs were not ported correctly.

The third problematic aspect of dpct comes when it has to deal with CUDA library calls. As a matter of fact, Crystal implementation uses extensively CUB library functions, for various tasks. dpct does not to port CUB function calls automatically, thus the software develper needs manually to replace these with calls to DPC++ functions that are semantically equivalent.

Eventually, in some cases, even when the DPC++ conversion is semantically correct, it might be suboptimal in terms of performance. An example is the call to the memory barrier function. dpct converts it automatically into a memory fence in both global memory and local memory which are very expensive. However, in this specific context, a memory fence in the local memory of each work-group was sufficient. Thus, we optimized the code generated by dpct manually and reimplemented CUDA atomic functions like atomicCAS and atomicAdd. Therefore, despite being a rather useful tool, dpct is still far from being an accurate porting tool from CUDA to SYCL.

3.3 Projection Design

We consider two forms of projection queries: one that computes a linear combination of columns (Q1) and one involving user defined function (Q2):

Q0 : SELECT $ax_1 + bx_2$ FROM R;
Q1 : SELECT $\sigma(ax_1 + bx_2)$ FROM R;

where x_1 and x_2 are 4-byte floating point values. The number of entries in the input array is 2^{29} . σ is the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

which can represent the output of a logistic regression model.

Following crystal's block-wide functions design, we implement a single kernel that does two block loads to load the tiles of the respective columns, computes the projection and eventually does a block store to store it in the result array.

3.4 Hash Join Design

The join operator comprises two kernels: a build kernel and a probe kernel. The first one populates the hash table with the tuples of the smaller, build relation. Following crystal's approach, we implement a linear probing strategy with the hash table being implemented as a simple array of slots with each slot containing a key and a payload without any pointers. The probe kernel uses the other relation to search for matches in parallel. Each thread block loads a tile from the probe table, and each thread computes the local sum for a subset of tile elements that meet the predicate condition. Then, all local values are aggregated in a hierarchical fashion, first for all threads within a block, and then across all thread blocks

3.5 FPGA Select Design

The design of the select for FPGA follows the best practices of intel oneAPI[3], in particular for its implementation we used **unrolled loops** and **predication**. By replicating the computational logic inside a loop, the loop unrolling process boosts software parallelism. Both partial and complete unrolling are possible. Unrolling loops is a typical technique in an FPGA design to directly trade off on-chip resources for greater performance. Although increasing the unroll factor speeds up throughput, it also causes increased resource usage, which finally exhausts the available memory (bottleneck). In our implementation, we discovered that using a fully enrolled loop offers the optimal trade-off.

```
for each y in R:
  if y > v:
    output[i++] = v
```

(a) With branching

```
for each y in R:
  output[i] = y
  i += (y > v)
```

(b) With predication

Figure 3: Branching vs Predication

An if-statement is used in the branching implementation to carry out the selection. The penalty for branch mispredictions is the fundamental issue with the branching implementation. The branch predictor is unable to forecast the branch outcome if the selectivity of the condition is neither too high nor too low. Performance is hampered as a result of pipeline stalls. Predication, instead, transforms the branch (control dependency) into a data dependency. Spatial implementations can communicate results backwards in the pipeline to work that began in a later cycle (i.e., to work at an earlier level in the pipeline) quite well. By sending outcomes backward in the pipeline, this enables effective data dependence communication. The ability to pass data backward (to an earlier stage in the pipeline) is key to spatial architecture [11]. The main approach that make expressing this pattern easy is through loops. When a spatial compiler implements a loop, iterations of the loop can be used to fill the stages of the pipeline.

3.6 FPGA Hash Join design

the design of Hash Join on FPGA exploits unrolled loops and **pipes**. In our implementation, we discovered that using a fully enrolled loop offers the optimal trade-off. The producer kernel receives the data from the host memory before sending it over a pipe to the join kernel, which builds the hash table and then performs the join over the probe table and eventually the consumer kernel writes the final result to the host memory. Figure 4 illustrates the high-level implementation design.

Figure 4: Design Hash Join

4 OPERATORS GPU VS CPU VS FPGA

In this section, we compare the performance of foundamental SQL operators - project, select, and hash join - on GPU, CPU and FPGA with the goal of understanding how the very same block-functions based implementation works on CPU, GPU, iGPU and how the custom implementation for FPGAs behaves. We will, also, extend the camprison to NVidia hardware targetted via DPC++ and hipSYCL [4].

4.1 Projection on CPU and GPU

In this section, we will show and comment the results of the projection operator on Intel's and NVidia's hardware, using the design described in **section 3.3**, with a two-columns table of size 2^{29}, which is the same study case presented in the original publication [12].

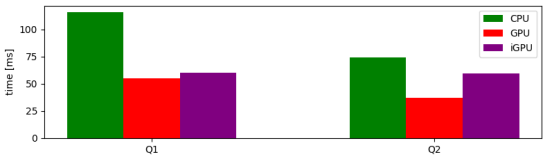

Figure 5: Project benchmarks on CPU, GPU and iGPU

Figure 5 shows the runtime of queries Q1 and Q2 on Intel's CPU, GPU and iGPU (shown as bars). The performance of Q1 is memory-bandwidth bound. GPU performs substantially better than the CPU implementation, due to its much higher memory bandwidth and slightly better than the iGPU.

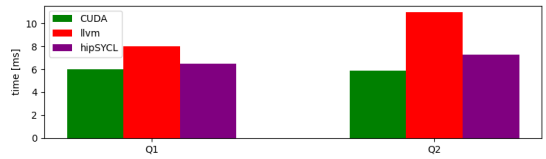

Figure 6: Project benchmarks on NVidia GPU

Figure 6 shows the results of the performed benchmarks using the very same setup but on NVidia GPUs, comparing hipSYCL, Intel's llvm and the original CUDA code.

We notice how for both the queries we achieve similar performances and the ported SYCL code is only 5% slower than the original CUDA code. In addition, hipSYCL yields slightly better performances than the cross-compiled code with llvm.

4.2 Hash Join on CPU and GPU

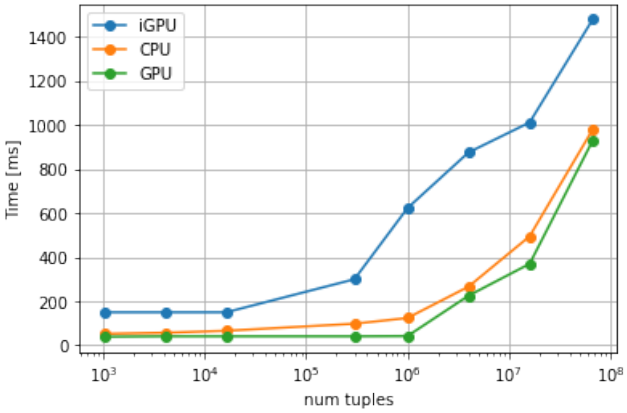

Figure 7: Hash Join benchmarks on CPU, GPU and iGPU

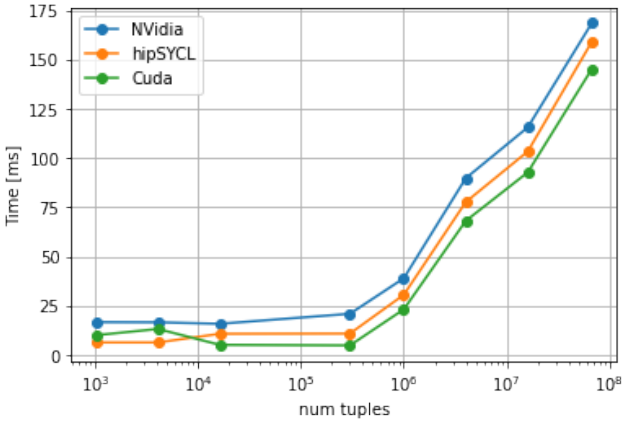

Figure 8: Hash Join benchmark on NVidia GPU

We study the Hash Join by focusing on the following microbenchmark query and configurations that are also used in the original Crystal publication and other prior literature[12].

```
SELECT SUM (A.v * B.v) FROM A, B WHERE A.k = B.k;
```

where tables A and B consist of two 4-byte integer columns k, v joined on key k. The size of the probe table is fixed at 256 million tuples, totaling 2GB of raw data. We use a hash table with 50% fill rate and vary the size of the build table such that it produces a hash table in the range 8KB-512MB. After fine-tuning the number of work-items-per-group, we fix this value to 128 both for the CPU and the GPU. Figure 7 shows the actual execution time of the probe kernel on Intel's CPU vs GPU vs iGPU for various build table sizes. Differently from prior work in Crystal, where only the probe time was benchmarked, we report the whole execution time, which includes the build time and the allocation time.

Looking at Figure 7, we see that the execution time increases twice, namely every time the hash table exceeds the size of the

cache (L2 and L3). for instance, for the CPU these thresholds are 256 KB and 12 MB.

Figure 8 demonstrates the comparison of the original Crystal CUDA implementation with the cross compiled version using Intel's llvm and the open-source hipSYCL toolchain, using a work-size of 256. Despite improving signifincatly with respect to Intel's discrete GPU, these results show that cross-compiled implementation is less efficient than the native implementation, which outperforms the cross-compiled one of 1.4 times so that there is room for further improvement.

However, considering the fact that the DPC++ implementation has the advantage of being executable on Intel® GPU and multicore CPU with no change in kernel code except for kernel dimensioning, we believe that trading off performance to achieve portability is one worth a serious consideration.

4.3 Evaluation of Operators in FPGA

Our experiments were conducted on Intel® DevCloud and we performs benchmarks of hash join and select operator on Intel® Arria and Intel® Strattix. We also calculated the Global Memory Bandwidth Usage. The LSU(Load Store Unit) bandwidth equation is the minimum of three bottlenecks we need to calculate the use of global memory bandwidth [1]. These formulas represent the theoretical maximum bandwidth an LSU may consume

$$\text{LSU bandwidth} = min\{BW_1, BW_2, BW_3\} \text{ MB/s} \quad (1)$$

$$\text{BW1} = k_{width} \cdot f_{max}$$

$$\text{BW2} = m_{width} \cdot f_{max}$$

$$\text{BW3} = \frac{\text{BW}_{\max} \cdot N_{ch}^{i}}{N_{ch}}$$

where k_{width} represent the byte-width of the LSU on the kernel, m_{width} represent the byte-width of the LSU facing the external memory, f_{max} is the clock-speed of the kernel in MHz, $\text{BW}_{\max}$ is the Maximum Bandwidth the global memory can achieve, N_{ch} represent the number of interfaces and external memory has and N_{ch}^{i} assume the same value of the number of channels if interleaving is enabled, otherwise is 1.

4.4 FPGA Select

We use the following micro-benchmark to evaluate selections:

```
SELECT y FROM R WHERE y < v;
```

where we vary the selectivity of the predicate from 0.1 to 1. The input table is composed of two columns of 32-bit integer, for our evaluation we generate a table of 2^{28} rows. The entire input array is scanned and written to the output array. The measured runtime - only for load/store from host memory to device memory and vice versa - can be computed as follows

$$runtime = \frac{8N}{B_r} + \frac{8N}{B_w}$$

where B_r represent the bandwidth for the load, B_w represent the bandwidth for the store while N is the number of rows. By looking at the report generated, it were used two LSUs for the load and two LSUs for the store. On Intel® Arria, the estimated Frequency is 240 MHz, therefore plugging into Equation 1, we end up with a measured bandwidth of 2224 MB/s for the load and 2224 MB/s for the store

Table 1 shows detailed results on the Total latency and kernel time.

Table 1: Evaluation on Intel Arria 10 GX

Selectivity	Total latency[ms]	kernel time[ms]
0.1	2786	967.8
0.2	2781.2	967.8
0.4	2762.6	967.8
0.6	2787.9	967.8
0.8	2786.4	967.8
1	2785.3	967.8

We conclude that 70% of the total latency is spent to load and store from the host memory to the device memory and vice versa.

Reproducing the very same benchmarks on Intel® Strattix FPGA board, where the estimated Frequency is equal to 432 MHz, using Equation 1, we obtain a measured bandwidth of 2872 MB/s for the load and 2872 MB/s for the store, so the time spent only for the load and store is 1495 ms, less than Intel® Arria but still corresponding to the 60% of the total latency. Table 2 shows detailed results of the total latency and kernel time.

Table 2: Evaluation on Intel Strattix 10 SX

Selectivity	Total latency[ms]	kernel time[ms]
0.1	2310	749.0
0.2	2316.66	749.0
0.4	2311.48	749.0
0.6	2311.66	749.0
0.8	2316.7	749.0
1	2298.66	749.0

For the sake of completeness, Table 3 also reports the results on the use of the available area.

Table 3: Area analysis

	ALUTs	RAMs	DSPs
Static Partition	179950 (21%)	492 (18%)	123 (8%)
Kernel System	5897 (1%)	89 (3%)	0 (0%)

4.5 FPGA Hash Join

In this section we will evaluate the performances of the DPC++ Hash join implementation running on Intel Arria® 10 GX and Intel Strattix 10 SX and then compare them with the previous benchmarks obtained on Intel CPU and discrete GPU. We use the following micro-benchmark to evaluate the operator:

```
SELECT R.k, R.v, S.k, S.v FROM R, S WHERE R.k = S.k;
```

where R and S are two-columns table of 32-bit integer. Due to kernel memory restrictions, we keep fixed the Build table to have size of 2 MB (2^{18} rows) and we vary the size of the the Probe table. The entire Probe table and Build table are read and then only the joined rows are written to the output array, thus having 3 columns (key, value_build_table, value_probe_table). The measured runtime, only taking into account load/store from host memory to device memory or vice versa, in the worst case scenario, is equal to

$$runtime = \frac{8P + 8B}{B_r} + \frac{12P}{B_w}$$

where B_r represent the bandwidth for the load, B_w represent the bandwidth for the store, P is the number of rows of the Probe table and B is the number of rows of the Build table. The produced report shows that 4 LSUs have been employed for the load and 3 LSUs for the store. Taking into that that the estimated frequency on Intel® Arria is equal to 240 MHz, using Equation 1, we obtain a measured bandwidth of 3536 MB/s for the load and 2652 MB/s for the store. Table 4 shows the results of the benchmarks on Intel® Arria.

Table 4: Evaluation on Intel Arria 10 GX

Probe Table size	Total latency[ms]	kernel time[ms]
8kB	46.7	1.01
32kB	45.98	1.05
128kB	45.67	1.00
3MB	47.8	2.39
8MB	51.2	4.76
32MB	85.2	19.03
128MB	222.29	76.10
512MB	805.6	304.3

We notice that, on average, 40% of the time is spent for load and store operations. Generating the report for Intel® Strattix instead, the estimated frequency is 349.92 MHz, so doing similar calculations as before , we obtain a measured bandwidth of 5392 MB/s for the load and 4044 MB/s for the store. Table 5 shows the results on Intel® Strattix.

Table 5: Evaluation on Intel Strattix 10 SX

Probe Table size	Total latency[ms]	kernel time[ms]
8kB	56	0.65
32kB	56.8	0.60
128kB	56.5	0.61
3MB	55.1	1.56
8MB	55.48	3.12
32MB	84.1	12.47
128MB	198.8	49.84
512MB	635.4	200

Similarly to Arria, we conclude that on average 40% of the time is spent performing load and store operations. Table 6 reports the area usage of the device.

Table 6: Area analysis

	ALUTs	RAMs	DSPs
Static Partition	466792 (25%)	3039 (26%)	1291 (22%)
Kernel System	29053 (2%)	1556 (13%)	0 (0%)

Finally, Figure 9 shows a summary plot comparison of the FPGAs benchmarks against Intel® Iris® Xe MAX Graphics (dGPU) and Intel® Xeon® Gold 6128 CPU.

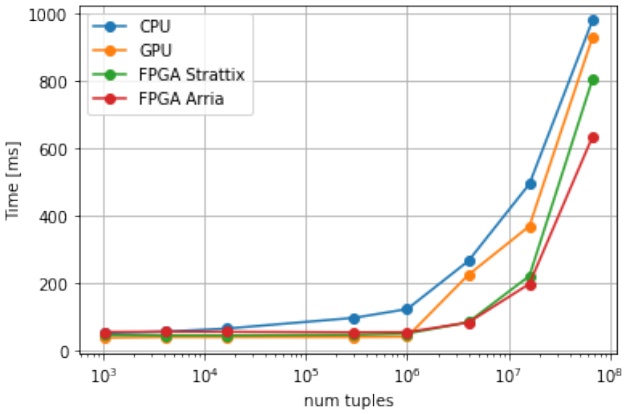

Figure 9: Benchmarks on FPGAs, GPU, CPU

5 WORKLOAD EVALUATION

Now that we have a good understanding of how individual operators perform on CPU, GPU and FPGA, we will evaluate the performance of a workload of full SQL queries on these hardware platforms. We first describe the query workload we use in our evaluation. We then present a high-level comparison of the performance of queries implemented with the tile-based execution model running on Intel's CPU, GPU and iGPU versus Nvidia GPU. We also report the performance of the original CUDA implementation of crystal. In addition, we propose a comparison with the performances yield on NVidia Hardware compiled with the open-source hipSYCL[4] runtime.

For experiments run on Intel Hardware, including FPGAs, we use Intel's devcloud platform. For experiments run on Nvidia GPU, we use an machine equipped with an Nvidia NVIDIA GeForce RTX 2080 Ti GPU, CUDA 11.4 and running on Ubuntu 20.04. In our evaluation, we ensure that data is already loaded into the respective device's memory before experiments start. We run each experiment 3 times and report the average measured execution time

5.1 workload

For the full query evaluation, we use the Star Schema Bench-mark (SSB) [10], a simplified version of the more popular TPC-H benchmark. It has one fact table lineorder and four dimension tables date, supplier, customer, part which are organized in a star schema fashion. There are a total of 11 queries in the benchmark, divided into 4 query flights. In our experiments we run the benchmark with a

scale factor of 20 which will generate the fact table with 120 million tuples. The total dataset size is around 13GB.

5.2 performance comparison

In order to ensure a fair comparison across systems, we dictionary encode the string columns into integers prior to data loading and manually rewrite the queries to directly reference the dictionary-encoded value, as done in crystal. For example, a query with predicate s_region = 'ASIA' is rewritten with predicate s_region = 2, where 2 is the dictionary-encoded value of 'ASIA'. However, in our benchmark we make sure all column entries are 4-byte values to ensure ease of comparison with other systems and avoid implementation artifacts. We use a thread block size of 128 with tile size of 4 (= 4×128) resulting in 4 entries per thread per tile. The results are shown in Figure 10. We can notice how the NVidia GPU significantly outperforms Intel's hardware with the same implementation, resulting on average in a 3.5x speedup with respect to the discrete GPU and a 5.8x speedup with respect to the CPU. Nevertheless, the original CUDA implementation of crystal shows on average a 1.9x speedup with respect to the SYCL porting, with the latter one outperforming the first one for some queries.

The further comparison with hipSYCL is shown in Figure 11. We can notice a speedup for smaller queries, but on average both llvm and hipSYCL show similar performances with just a 0.01 mean discrepancy.

6 CONCLUSION AND FUTURE WORKS

Developing applications that are performance-portable has been a major challenge in the HPC world, and we believe that lack of performance portability is one of the main reasons hindering a much broader adoption of XPU by data management systems. Our work shows that single-source, cross-architecture programming models like DPC++ are a step in the right direction, which opens up several other lines of future research.

On the runtime front, more work is required to understand the gap in performance between DPC++ and other optimized, proprietary platforms like CUDA.

reffig:fpga.

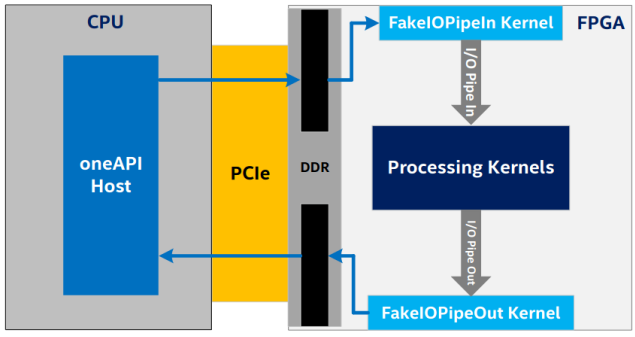

Figure 12: FPGA architecture

On the FPGA front, we conducted an analysis aiming at comparing its performances with CPUs and GPUs. Based on these analysis, we showed that major obstacle for accelerating DBMS operators on the FPGA is the memory bandwidth as also pointed out by [5]. The widely used CPU-FPGA platforms are usually discrete, in which an FPGA board with private memory resource is attached via PCIe bus as a peripheral of the CPU, as shown in Figure The compiler instantiates a specialized LSU for each access site based on the memory access pattern to maximize the efficiency of data accesses. All accesses to global memory must go through the hardware interface. The compiler connects every LSU to an existing hardware interface through which it transacts with device global memory. Since the compiler cannot alter that interface or create more such interfaces, it must share the interface between multiple datapath reads or writes, which can limit the throughput of the design [1]. The strategies used by the compiler to maximize efficient use of available memory interface bandwidth include (but are not limited to) the following:

- Eliminating unnecessary accesses.
- Statically coalescing contiguous accesses.
- Generating specialized LSUs that can perform the following:
  - Dynamically coalesced accesses that fall within the same memory word (as defined by the interface)
  - Prefetch and cache memory contents

In this architecture, before any computation on the FPGA can start, the CPU must copy the data to the FPGA's private memory through the PCIe bus and copy the results back after the computation is finished. Therefore, data transport and/or synchronization overhead typically place limits on speed improvements. Tightly coupled CPU-FPGA designs have been developed to pro- vide memory coherency between CPU and FPGA in order to reduce communication latency and reduce data transmission overhead. For example, Intel started the Hardware Accelerator Research Program (HARP)[2]. So it would be interesting in the future to explore this Architecture and repeat the same benchmarks and comparing it with a stateof-the-art FPGA-based join implementation[7]. However, FPGAs are a cheaper off-the-shelf alternative that you can reprogram for each new application, while a custom ASIC even if generally outperforms an FPGA on a specific task, they take significant time and money to develop.

REFERENCES

[1] [n.d.]. https://www.intel.com/content/dam/develop/external/us/en/documents/oneapi-dpcpp-fpga-optimization-guide.pdf
[2] [n.d.]. https://cpufpga.files.wordpress.com/2016/04/harp_isca_2016_final.pdf.
[3] [n.d.]. oneAPI-samples/DirectProgramming/DPC++FPGA/ReferenceDesigns/db at master · oneapi-src/oneAPI-samples — github.com. https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2BFPGA/ReferenceDesigns/db. [Accessed 27-Jun-2022].
[4] Aksel Alpay and Vincent Heuveline. 2020. SYCL beyond OpenCL. In *Proceedings of the International Workshop on OpenCL*. ACM. https://doi.org/10.1145/3388333.3388658
[5] Xinyu Chen, Yao Chen, Ronak Bajaj, Jiong He, Bingsheng He, W. Wong, and Deming Chen. 2020. Is FPGA Useful for Hash Joins?. In *CIDR*.
[6] Tom Deakin and Simon McIntosh-Smith. 2020. Evaluating the Performance of HPC-Style SYCL Applications. In *Proceedings of the International Workshop on OpenCL* (Munich, Germany) *(IWOCL '20)*. Association for Computing Machinery, New York, NY, USA, Article 12, 11 pages. https://doi.org/10.1145/3388333.3388643
[7] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR*.
[8] Eugenio Marinelli and Raja Appuswamy. 2021. XJoin: Portable, Parallel Hash Join across Diverse XPU Architectures with OneAPI. In *Proceedings of the 17th*

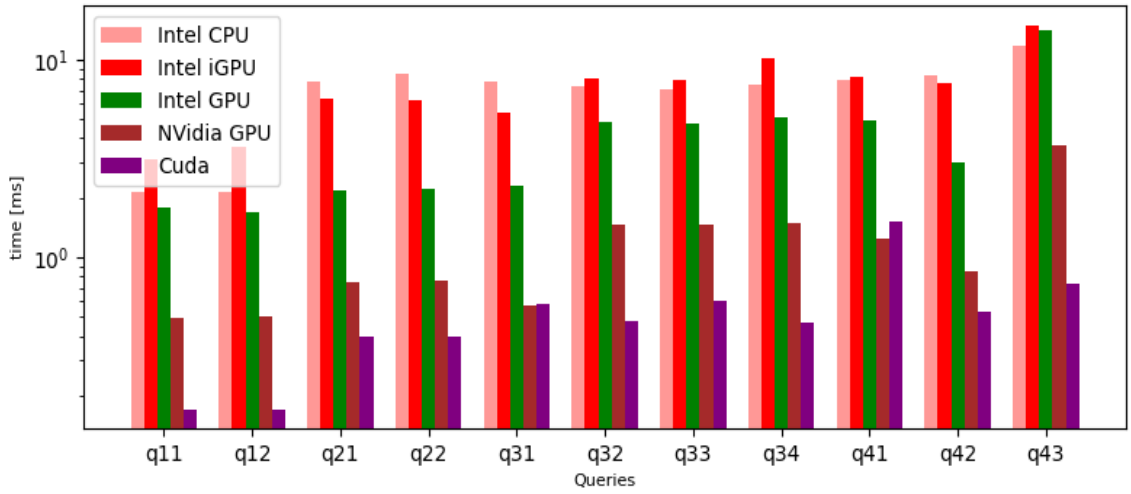

Figure 10: Star Schema Benchmark Queries

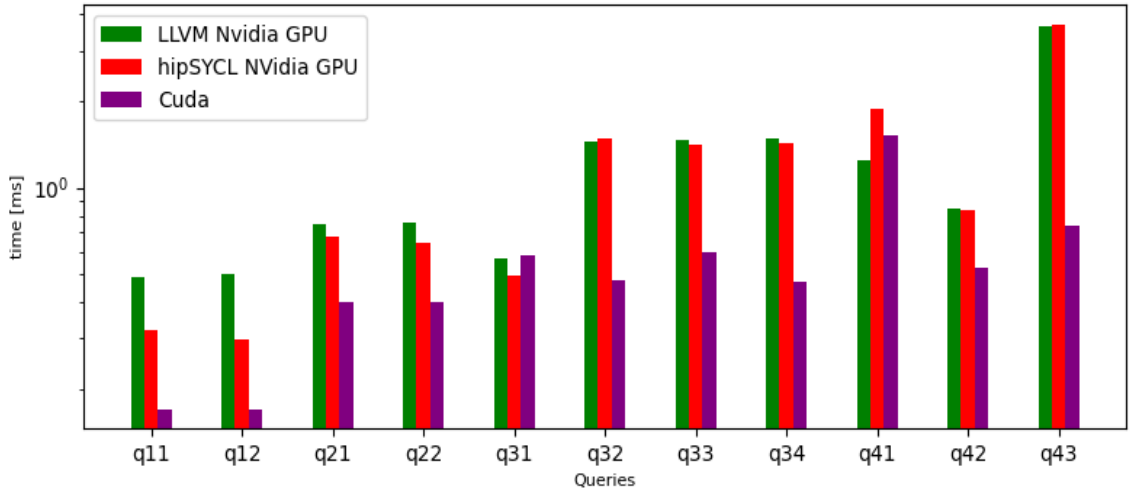

Figure 11: Star Schema Benchmark with hipSYCL

International Workshop on Data Management on New Hardware (DaMoN 2021) (Virtual Event, China) *(DAMON'21)*. Association for Computing Machinery, New York, NY, USA, Article 11, 5 pages. https://doi.org/10.1145/3465998.3466012

[9] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[10] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking*, Raghunath Nambiar and Meikel Poess (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 237–252.

[11] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian. 2020. *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress. https://books.google.fr/books?id=vLI7zAEACAAJ

[12] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. 2020. 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1617–1632. https://doi.org/10.1145/3318464.3380595