XJoin: Portable, parallel hash join across diverse XPU architectures with oneAPI

Eugenio Marinelli Raja Appuswamy
EURECOM
Biot, France
firstname.lastname@eurecom.fr

ABSTRACT

Modern server hardware is increasingly heterogeneous with a diverse mix of XPU architectures deployed across CPU, GPU, and FPGAs. However, till date, database developers have had to rely on either proprietary, architecture-specific solutions (like CUDA), or low-level, cross-architecture solutions that complicate development (like OpenCL). The lack of portable parallelism caused by the absence of a common high-level programming framework is one of the main reasons preventing a wider adoption of XPUs by 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, we port a recently-proposed, highly-optimized, GPU-based hash join algorithm from CUDA to Data Parallel C++ (DPC++). We then execute the hash join on multicore CPUs, integrated GPUs (Intel GEN9), and discrete GPUs (Intel DG1 and NVIDIA GeForce) without changing a single line of kernel code to demonstrate that DPC++ enables portable parallelism. We compare the performance of DPC++ kernels with hand-optimized CUDA kernels and model-based theoretical performance bounds to demonstrate the performance–portability trade off in using DPC++.

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. However, a major factor that has limited the wide-spread adoption of XPU in database systems has been the lack of portable parallelism. Historically, general-purpose, standardized programming languages like C++ were developed before the evolution of heterogeneous parallel computing. Thus, there was no common software stack for programming XPU, and developers had to use vendor-specific programming platforms and APIs. This problem was particularly acute in High-Performance Computing (HPC), where limited compiler support for a particular combination of architecture and programming model forced HPC developers to maintain multiple versions of codes in each programming model. The need for a portable programming model that enables just one version of the source code to work across diverse architectures led to the development of OpenCL, a cross-industry framework initiative that provided a "C"-like language for writing compute-intensive kernel that can be offloaded onto any supported XPU using a runtime API.

Over the past few years, HPC applications have evolved to adopt more general-purpose programming languages like C++ instead of C and FORTRAN, and HPC installations expanded to adopt XPU from more vendors. This change exposed several limitations of low-level frameworks like OpenCL. First, the low-level nature of OpenCL was meant to directly expose data parallelism in underlying hardware while leaving everything else from data movement to kernel dispatch to developers leading to boilerplate code verbosity. Second, programs written in OpenCL are not single-source in nature as kernel code needs to be separated from host code, represented as strings and separately managed complicating software development. These challenges led to the development of custom HPC frameworks like RAJA [2] and Kokkos [5] that bridge the gap by providing C++ abstraction layers for portable parallel execution. Thus, in turn, spurred the development of SYCL, an open, industry-standard, single-source, modern C++ parallel programming model from Khronos group (who also maintain OpenCL).

Recent work has investigated the performance–portability trade offs in using SYCL for accelerating key HPC applications [4]. In this work, we investigate the utility of SYCL in the development of performance-portable database engines by focusing on hash join. First, we start from a state-of-the-art, data-parallel hash join that has been developed in CUDA and optimized for execution on NVIDIA GPU. We port the join to Data-Parallel C++ (DPC++)–an open-source implementation of SYCL–using the oneAPI toolkit. We refer to the DPC++-based hash join as *XJoin* in the rest of this paper. We then execute XJoin on Intel® multicore CPU, integrated Intel® GEN9 GPU, Intel® Iris® Xe Max DG1 discrete GPU, and NVIDIA RTX 2080 Ti discrete GPU, without changing the data-parallel kernel code to demonstrate cross-architecture, cross-vendor portability of DPC++. Using models that provide theoretical upper bounds for CPU performance, and using the state-of-the-art, hand-optimized CUDA join, we investigate the performance of XJoin to understand the effectiveness of DPC++ in parallelizing on CPU and GPU. We make the XJoin source code publicly available[1] to encourage further work on performance-portable database engines.

DAMON'21, June 20–25, 2021, Virtual Event, China
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8556-5/21/06…$15.00
https://doi.org/10.1145/3465998.3466012

[1]https://github.com/Eug9/XJoin

2 DESIGN

Our goal in this work is to take the first steps towards investigating the utility of DPC++ in developing performance-portable database engines. In order to do so, we focus on the hash join operator as it has been extensively studied in the database literature, with many different hash join algorithms proposed for both CPU and GPU [3, 12]. In this work, we build on a recent, state-of-the-art hash join from the Crystal library that has been optimized for data-parallel execution on GPU [13]. In this section, we first provide an overview of the Crystal GPU join. Then, we detail how we ported it from CUDA to DPC++.

2.1 Crystal GPU Join

The novelty of Crystal lies in its tile-based implementation strategy. 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 shared memory and synchronize through barriers. The set of data elements that can be collectively processed by a thread block is referred to as a tile. The basic compute unit in Crystal is a tile, which is a sub slice of the input data. This approach makes it possible to write kernels in terms of block-wide functions that take work with a set of tiles as units of input and output. Each function uses vector instructions for memory accesses, and registers for storing values.

Using block-wide functions, Crystal implements a no partitioning join, which uses a non-partitioned global hash table. The join operator comprises two kernels, a build kernel and a probe kernel. The build kernel populates the hash table with the tuples of the smaller, build relation. Crystal implements a linear probing strategy due to its simplicity, 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.

2.2 DPC++ conversion

In order to port the Crystal join from CUDA to DPC++, we start with tooling support from oneAPI[2]. oneAPI is a cross-industry effort for developing an open, standards-based unified programming model to simply software development across diverse accelerator architectures. In addition to providing an open-source implementation of SYCL with DPC++, Intel® oneAPI also provides performance-portable, hardware-accelerated libraries to enable API-based programming for various popular application domains, and tools to assist in developing and profiling DPC++ applications. In this work we focus on DPC++ Compatibility Tool that aims to convert CUDA code to DPC++ at syntax level, recognizing the main CUDA constructs and converting them to their DPC++ equivalent. We use this tool to convert the Crystal hash join implementation together with necessary block-wide functions [13] from CUDA to DPC++. 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++, in order to assist in future migration of current CUDA-based GPU database engines [8–11]

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. 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. The compatibility tool converts the two CUDA join kernels - build and probe - in two DPC++ *nd_range parallel_for* kernels, and automatically adds the *id_item* as parameters of all functions called in the kernel code. 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. For example, in the original code, threads in the probe-kernel have to compute the sum of the product for all entries that match the query predicate. This involves a certain number of local sum computations performed by each thread that are first aggregated at the tile level by all threads within a thread block, and then aggregated across all thread blocks. This involves the use of memory barriers, atomics, and synchronisation at various kernel execution stages. More precisely, all threads in a warp compute aggregate their value using a low level primitive (*shuffle_down*) that allows inter-thread communication without any cost. The value computed by each warp is saved in local memory. A tree-reduction pattern is used to compute the aggregate sum per thread block. Finally, after all thread blocks compute their local sum, the global sum is computed using atomic instructions in the global memory.

While *dpct* is able to convert the memory barrier and the atomic variables from CUDA to DPC++, it was not able to replace the warp-level functions which are a central piece of the Crystal tile-based probe kernel. Thus, we had to reimplement the logic. DPC++ already

[2]www.oneapi.com

provides a set of functions that implement the main data-parallel patterns at the work-group level. Thus, we map the concept of a tile from Crystal to a work-group in DPC++ and use the *reduce()* function of the *work_group* class to perform tile-level reduction directly without having to implement warp-level shuffles and block-level tree reduction manually.

The third problematic aspect of *dpct* is with respect to library calls. For instance, the original CUDA implementation uses extensively CUB library functions, for various tasks. *dpct* does not to port CUB function calls automatically. As work around, we manually replace these with calls to DPC++ functions that are semantically equivalent.

Finally, 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 case, a memory fence in the local memory of each work-group was sufficient. Thus, we optimized the code generated by *dpct*.

3 EVALUATION

In this section, we will present the experimental results. First, we will investigate the ability of DPC++ to effectively parallelize XJoin on multicore CPUs. Then, we will investigate the cross-architecture portability of DPC++ by presenting a comparison of XJoin running on Intel® CPU, GEN9 iGPU and the recently-released Intel® dGPU. Finally, we will investigate the cross-vendor portability of DPC++ by comparing cross-compiled XJoin with the native CUDA implementation of Crystal join using an NVIDIA dGPU.

3.1 Experimental Setup

3.1.1 Hardware Setup. We evaluate the DPC++ implementation on two servers from Intel® DevCloud. The first one is equipped with an Intel® GEN9 iGPU and a 6-core Intel(R) Xeon(R) E-2176G CPU clocked at 3.70GHz; the second one with a Intel® Iris Xe Max DG1 dGPU. For the tests where we compare Crystal's native CUDA hash join with cross-compiled (DPC++ to CUDA) XJoin, we use a local server equipped with an NVIDIA GeForce RTX 2080 Ti dGPU.

3.2 Scalability on CPU

In this section we present the results obtained by executing XJoin on a 6-core CPU. 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 [1].

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

Tables A and B consist of two 4-byte integer columns *k*, *v*. The two tables are joined on key *k*. The size of the probe table 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.

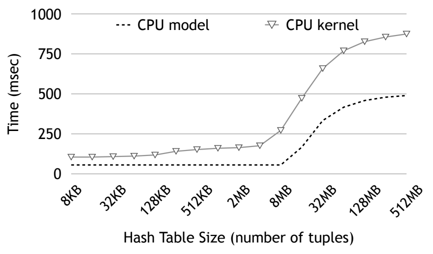
Figure1a shows the actual execution time of the probe kernel on CPU for various build table sizes. We only report the execution time for the probing phase of the join similar to prior work in Crystal [13] to save space, as the probe table is much larger than the build table, and the build kernel takes a fraction of time of the probe. In addition, in order to understand how well XJoin performs on the CPU, we also show the lower bounded execution time obtained from a theoretical model that was also used in Crystal evaluation [13]. The theoretical approach only considers the probe phase and models the runtime such that the kernel is bounded by the device memory bandwidth and/or by the cache bandwidth depending on whether the hash table fits into one of caches available or not.

Note that the model provides a theoretical lower bound achievable with the CPU as it assumes the program can exploit the maximum memory bandwidth. In practice, the probe phase involves many random memory accesses. As a result, observed memory bandwidth is often much lower than peak bandwidth. Looking at Figure1a, we see that the execution time increases twice, once after 256KB when the hash table size does not fit in the L2 cache, and once after 12MB when the hash table exceeds the size of the L3 cache. We also see that in all cases, XJoin performance is within $2\times$ of the theoretical model. This shows that the DPC++ runtime is able to effectively parallelize the data-parallel kernels, which were originally designed for the GPU, to also exploit multicore CPU.

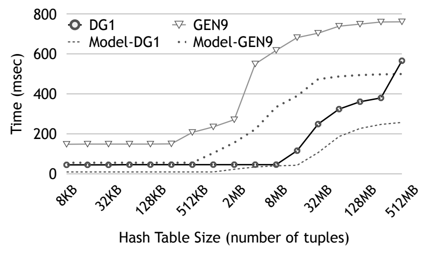
Right-sizing the kernel by specifying the right work-group size plays a key role in achieving this CPU performance. In order to understand the sensitivity of performance with respect to kernel sizing, we test different values for the work-group size. Figure 2a shows the performance of XJoin on the CPU as we vary the work-group size from 64 to 2048. We fixed the probe table size at 256M tuples and the hash table size at 4KB for this experiment. This result clearly shows that similar to GPU kernels, the optimal choice of work-group size on the CPU plays an important role in performance. The results shown in Figure 1a are based on a work-group size of 1024 as it provides best performance.

3.3 XPU scalability

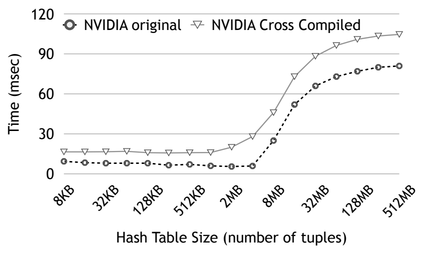
In this section we will show performances of the DPC++ join implementation running on Intel® DG1 dGPU and GEN9 iGPU, and compare these results with the previous CPU experiment. Similar to the CPU case, work-group sizing plays an important role in GPU cases also. However, the maximum number of possible work-groups varies across devices. For both GPUs, we vary the number of work-items in each work-group from 64 up to their limit. We fix the probe table size at 256M tuples and the hash table size at 4KB. Figures 2b, 2c show the execution time of the probe kernel in various settings on GEN9 iGPU and DG1 respectively. As we get the best performance with 128 work-items per group for both GEN9 and DG1, we use those for sizing kernels. Using optimal work-group sizes, we compare the performance of different architectures by varying the size of hash table while keeping the probe table size fixed at 256M tuples. Figure 1b shows execution time of probe kernel on dGPU and iGPU. As expected, the execution time increases when the hash table size does not fit in the cache. For the iGPU, these increases happen after 512KB (L2) and 2MB (L3). For DG1, performance drop is observed after 64KB (local) and 20MB (LLC). Figure 1b also shows the execution time obtained from the theoretical model. XJoin lags the theoretical results by up $5\times$ under DG1. Comparing Figures 1b,1a, we also see that the DG1 dGPU outperforms both GEN9 iGPU and the 6-core CPU. DG1 has 96 execution units compared to 24 in GEN9 iGPU. DG1's onboard global memory has a bandwidth of 62 GBps compared to the host DRAM's bandwidth of 35 GBps. As the probe table is fixed at 2GB,



(a) CPU execution time.

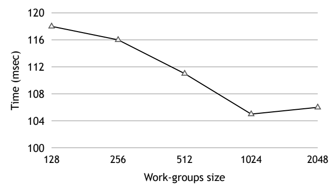


(b) XJoin exec. time on CPU, iGPU, and dGPU.

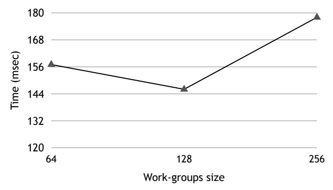


(c) Exec. time of XJoin vs Crystal.

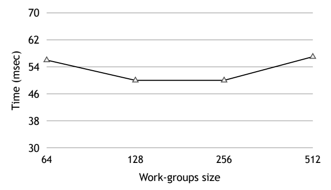
Figure 1: Execution time of XJoin probe kernel in various settings.



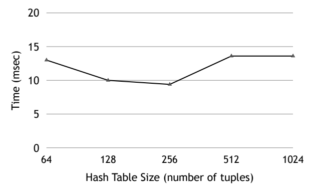
(a) CPU fine-tuning



(b) GEN9 fine tuning



(c) DG1 fine tuning



(d) NVIDIA GPU fine tuning

Figure 2: Fine-tuning kernel dimension parameter.

it fits completely in the global memory of DG1 leading to better performance than CPU/iGPU, and a bigger deviation of the theoretical model from the observed execution time. It is important to note here that aside from the change in kernel dimensioning, and choice of backend device, there was no change to the core kernel source code. Thus, these results clearly demonstrate the cross-architecture portability of DPC++.

3.4 Cross-platform execution

We will now demonstrate the cross-platform portability of DPC++ by comparing the performance of XJoin with respect to the original Crystal hash join using NVIDIA GPU. In order to run XJoin on NVIDIA GPU, we used CodePlay's SYCL-for-CUDA extension[3] that allows compiling applications written in DPC++ to run on NVIDIA dGPUs. In terms of code, the main change required is the recompilation of XJoin with a modified Clang++–LLVM compilation infrastructure that supports a CUDA backend.

As before, Figure 2d shows the sensitivity of XJoin to work-group size on the NVIDIA GPU. Figure 1c shows the execution time of the probe kernel using the empirically-estimated optimal work-group size of 256. Comparing Figures 1c, 1b, we see that XJoin on NVIDIA GPU outperforms the DG1 by up to 5.4× when the hashtable does not fit in last-level cache. This is expected given that the global memory on the NVIDIA GPU has a bandwidth of 616 GBps compared to the 62GBps of DG1. Figure 1c also shows the execution time of the original, hand-tuned, CUDA-based Crystal hash join. Note that both joins have an inflection point in performance beyond 8MB, as the size of the L2 cache in the GPU is 5.5MB. Comparing XJoin and Crystal join, we see that the cross-compiled, DPC++-based XJoin is always slower than its CUDA-based counterpart. The worst case difference between the two is at 4MB, when the hashtable fits in the L2 cache, where XJoin is 4.7× slower than Crystal join. However, beyond 8MB, the probe table accesses are served from global memory, and the difference between the two drops to 1.29×. These results show that cross-compiled implementation is less efficient than the native implementation and 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.

[3]https://github.com/codeplaysoftware/sycl-for-cuda

4 CONCLUSION AND FUTURE WORK

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 as they will enable key data-parallel kernels to be written using standard C++ and coexist with other components. Our work opens up several other lines of future research. On the XPU front, an immediate avenue of future work is evaluating XJoin on FPGA and comparing it with a state-of-the-art FPGA-based join implementation [7]. On the runtime front, more work is required to understand the gap in performance between DPC++ and other optimized, proprietary platforms like CUDA. On the database architecture front, an interesting avenue of future work is to implement vectorized primitives as data-parallel kernels in DPC++, with appropriately parameterized vector size, to enable performance-portable query execution not just on CPU but also on other XPU. Such an approach can be combined with traditional volcano exchange [6] to achieve intra-query heterogeneous parallelism, as a CPU-based exchange operator can coordinate the execution of different DPC++ primitives, and hence different parts of a pipeline, across different XPU.

REFERENCES

[1] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*.

[2] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *P3HPC*.

[3] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. 2014. *GPU-Accelerated Database Systems: Survey and Open Challenges*. Vol. 8920. 1–35.

[4] Tom Deakin and Simon McIntosh-Smith. 2020. Evaluating the Performance of HPC-Style SYCL Applications. In *IWOCL*.

[5] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216.

[6] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*.

[7] R. Halstead, Ildar Absalyamov, W. Najjar, and V. Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins. In *CIDR*.

[8] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. 2013. Hardware-Oblivious Parallelism for in-Memory Column-Stores. 6, 9 (2013).

[9] Kinetica. [n.d.]. https://www.kinetica.com.

[10] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. 2016. HippogriffDB: Balancing I/O and GPU Bandwidth in Big Data Analytics. 9, 14 (2016).

[11] Omnisci. [n.d.]. https://www.omnisci.com.

[12] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD*.

[13] 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)*. 1617–1632. https://doi.org/10.1145/3318464.3380595