OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures

Shuhao Zhang, Jiong He, Bingsheng He
Nanyang Technological University

Mian Lu
A*Star IHPC, Singapore

ABSTRACT

Driven by the rapid hardware development of parallel CPU/GPU architectures, we have witnessed emerging relational query processing techniques and implementations on those parallel architectures. However, most of those implementations are not portable across different architectures, because they are usually developed from scratch and target at a specific architecture. This paper proposes a kernel-adapter based design (OmniDB), a portable yet efficient query processor on parallel CPU/GPU architectures. OmniDB attempts to develop an extensible *query processing kernel* (qKernel) based on an abstract model for parallel architectures, and to leverage an architecture-specific layer (*adapter*) to make qKernel be aware of the target architecture. The goal of OmniDB is to maximize the common functionality in qKernel so that the development and maintenance efforts for adapters are minimized across different architectures. In this demo, we demonstrate our initial efforts in implementing OmniDB, and present the preliminary results on the portability and efficiency.

1. INTRODUCTION

We have witnessed the trend of heterogeneity in the development of parallel processor architectures. Hardware vendors have continued to offer different new multi-core/many-core processors. AMD and Intel offer multi-core CPUs, usually with fewer than eight cores. Sun Niagara and Cell processors have dozens of cores per chip. AMD and NVIDIA offer GPUs (Graphics Processing Units) that consist of dozens to hundreds of cores in a single chip. Traditionally, GPUs are usually connected with CPUs with PCI-e bus. Recently, coupled architectures (such as AMD APU (Accelerated Processing Units) and Sandy/Ivy bridge) integrate a CPU and a GPU into the same chip. How data management systems can fully leverage those architectures is still largely an open problem, and has attracted a lot of fruitful research efforts [1, 8]. Various architecture-aware query processors such as C-Store [16], GPUQP [9] and StagedDB [7] as well as query processing techniques (e.g., [2, 4, 12, 3, 11, 10]) have been developed. However, most of those implementations are developed from scratch, and they are usually tuned, optimized, and can only run on some specific architectures. For example, GPUQP can run on NVIDIA GPUs but cannot run on AMD GPUs. This creates code bases with similarities and differences. As time goes by, more parallel CPU/GPU architectures appear and more code bases would be created. Ideally, a portable and efficient query processor should be developed on different architectures. That motivates us to investigate the portability and efficiency of query processing on parallel CPU/GPU architectures.

Let us first discuss the pitfalls of having many code bases of query processors on different parallel CPU/GPU architectures, mainly from software engineering's perspective. One observation on architecture-aware query processors (e.g., [16, 9, 7]) is that some components can be shared among them, e.g., query parser. The other observation is that many of previous studies on architecture-aware techniques (e.g., [2, 4, 12, 3, 11]) have implicitly/explicitly made the assumption that they can be integrated into an existing query processor. Those observations lead to the necessity of developing and maintaining different code bases of query processors on different architectures. Even worse, those code bases may have common or similar components, and intersect with each other. As the hardware evolves, the code bases and their relationship need to evolve as well. According to the Laws of software evolution by Lehman [14, 15], the complexity of such software evolution is significantly increased. It requires a significant amount of work to maintain the software.

Portability is a must for reducing the development and maintenance cost. On the other hand, the efficiency of a query processor should be optimized according to the target architecture. To capture the best of both words, we propose a kernel-adapter based design (OmniDB), a portable yet efficient design for query processing on parallel CPU/GPU architectures. OmniDB attempts to develop an extensible *query processing kernel* (namely qKernel) based on an abstract model for parallel architectures, and to leverage a software layer (*adapter*) to make qKernel be aware of the target architecture.

It is an open problem on defining the boundary between qKernel and adapter so that software development and maintenance cost are minimized. Ideally, OmniDB should maximize the common functionality in qKernel to reduce the efforts on adapters. qKernel should be extensible to allow developers to plug architecture-specific parameters and

Proceedings of the VLDB Endowment, Vol. 6, No. 12

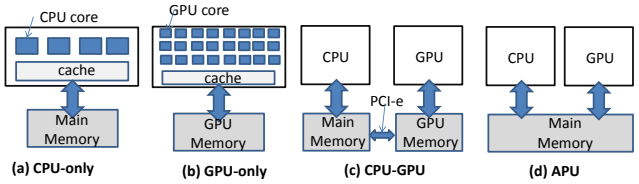

Figure 1: Example parallel architectures with one CPU and one GPU.

configurations into query processing operations and components. On the other hand, adapters include architecture-aware configurations, tunings and optimizations for qKernel.

It is not the intention of this demo to entirely solve the above-mentioned problem. Instead, we present our initial efforts in implementing OmniDB. This demo demonstrates a case for feasibility of the OmniDB design, and sheds some lights on defining the boundary. Our implementation is based on OpenCL (Open Computing Language) so that OmniDB can run on different architectures like CPUs and GPUs. We further develop adapters for four kinds of architectures: CPU-only, GPU-only, (classic) CPU-GPU and APU (We follow AMD's terminology). This demo presents the preliminary results on the portability and efficiency of OmniDB. First, OmniDB has the "one code base fits all" feature, whose code base can be compiled and run on all the four architectures. Second, we evaluate the effectiveness of adapters on different architectures. Our adapters effectively capture the differences among architectures.

2. BACKGROUND ON PARALLEL ARCHITECTURES

We currently focus on parallel architectures including multi-core CPUs and GPUs. Figure 1 illustrates the abstract view of four example parallel CPU/GPU architectures. This figure illustrates a machine with one CPU and/or one GPU for simplicity. The system design in this paper can be applicable to multiple CPUs and GPUs in the same machine. According to heterogeneity, we can divide them into two categories. CPU-only and GPU-only are homogeneous, whereas CPU-GPU and APU are heterogeneous. We briefly compare the similarities and differences among architectures within each category. More technical details have been elaborated in [6, 5].

As illustrated in Figures 1(a) and 1(b), both the CPU and the GPU are multi-/many-core architectures with a shared data cache. A GPU can have much more cores as well as much higher memory bandwidth than a multi-core CPU. On the other hand, the CPU has much larger L2/L3 data caches. Ideally, the GPU is more suitable for fine-grained data parallelism, and the CPU is more suitable for coarse-grained parallelism.

As illustrated in Figures 1(c) and 1(d), the CPU-GPU and the APU architectures utilize both the CPU and the GPU in the system (i.e., co-processing). The major difference is on how the CPU and the GPU communicate with each other. On the CPU-GPU architecture, the CPU and the GPU are connected with a low-bandwidth PCI-e bus. Thus, we need to carefully minimize the data transfer on the PCI-e bus. On the APU, the CPU and the GPU share the main memory directly and the PCI-e bus is eliminated. We have more flexibility in fine-grained co-processing.

3. DESIGN AND IMPLEMENTATION

In this section, we give an overview of the design of OmniDB, followed by our initial implementation. The design goal of OmniDB includes two aspects: portability and efficiency. We aim at designing a portable query processor that requires a minimum amount of efforts in achieving architecture-awareness for efficiency.

3.1 Architectural Design of OmniDB

Abstract architecture model. We model a parallel CPU/GPU architecture with N threaded-parallel processing elements (PPEs) P_1, ..., and P_N. Each PPE has its own memory space, which can be overlapped or non-overlapped with other PPEs. The memory accesses are in blocks. For example, we can model the CPU and the GPU as individual PPEs. On the APU, the two PPEs can share the main memory, whereas the CPU and the GPU have their own memory in the classic CPU-GPU architecture.

Our abstract architecture model is general for parallel query processing. It is able to capture a machine with multiple CPUs and GPUs, in which a PPE can be a CPU or a GPU. Also, our model is similar to PRAM (Parallel Random-Access Machine). Differently, the PPEs of our model may or may not share the main memory, whereas those of PRAM do.

A kernel-adapter based approach. To balance the portability and efficiency of architecture-aware query processing, we propose a kernel-adapter based approach to develop OmniDB. OmniDB consists of a query processing kernel (qKernel) and architecture-aware adapters.

Figure 2 illustrates the overall system design of OmniDB. Based on the abstract architecture model, qKernel consists of an execution engine, a scheduler, a cost model and other components in a standard query processor (such as query optimizer). There are some parameters and configurations in those components that will be customized by the specific adapter. The execution engine includes the data-parallel implementations for query processing operators. A workload scheduler is developed to assign *work units* to individual PPEs. The work unit is defined to a certain amount of work assigned to an PPE in one scheduling decision. In practice, it can be evaluating a query, an operator or processing a number of tuples. An abstract cost model is developed for estimating the execution cost. We estimate the total cost of executing a work unit on a PPE to be the total time of memory accesses and instruction executions.

An adapter includes the software components, parameters and configurations that adapt qKernel to the target architecture. It also instantiates the abstract architecture model to the target architecture specification (e.g., the cache size and the number of PPEs).

As discussed in Introduction, it is an open problem to define the boundary between qKernel and adapters. It is our long-term goal to solve this problem. In the next sub-section, we present our initial efforts in implementing OmniDB.

3.2 Preliminary Implementation

As a start, we use OpenCL to implement OmniDB. OpenCL is a framework for writing programs that execute across heterogeneous platforms such as CPUs and GPUs.

Execution engine. Like GPUQP [9], we adopt a layered design for the execution engine of OmniDB. This layered

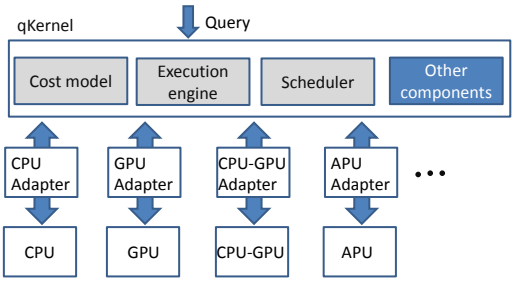

Figure 2: The kernel-adapter design of OmniDB.

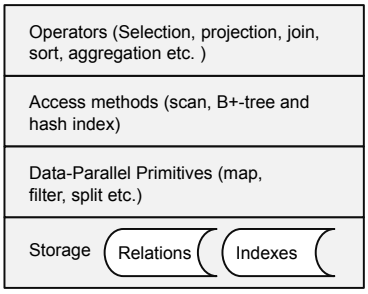

Figure 3: The layered design of execution engine.

design has high flexibility in software development and maintenance. Particularly, the execution engine consists of four layers from bottom up, including storage, data-parallel primitives, access methods and relational operators. Figure 3 shows the layered design. Primitives are common operations on the data or indexes in the system. The engine also supports common access methods, including the table scan, the B+-tree and the hash index, as well as a set of common query operators. Our access methods and relational operators are developed based on primitives. The data parallelism of those operations fits well on the abstract architecture model.

As a start, we develop the execution engine based on the code of GPUQP [9]. Most of the algorithmic design of primitives, access methods and operators can be found in the previous paper [9]. Beyond that, we have made the following major efforts.

First, we have ported GPUQP from CUDA to OpenCL, which allows the code to run on both multi-core CPUs and GPUs. Additionally, we modified the implementations of primitives, access methods and operators so that they can be customized to different work unit sizes. Note, GPUQP adopts fixed and fine-grained work unit size to take advantage of massive parallelism of the GPU.

Second, we have integrated more recent architecture-aware techniques into the execution engine, for example, tree index [13], sort and hashing [12]. Thanks to the kernel-adapter based design, we implement those algorithms into one code base (qKernel), rather than multiple code bases.

Third, we have implemented a new scheduling algorithm. GPUQP currently uses FIFO, which is suboptimal for CPU-GPU and APU architectures. The new scheduling algorithm considers multiple PPEs and the capabilities of PPEs.

Scheduler. The scheduling algorithm design has two major considerations. First, it balances the workload among different PPEs (i.e., avoiding the contention). Second, query processing performance may vary on different PPEs, and the scheduling algorithm prefers to choose the PPE that achieves the best performance.

Since calculating the optimal scheduling plan for a workload is an NP problem, we adopt an on-line greedy algorithm. Particularly, our scheduling algorithm chooses the suitable PPE as follows. First, we estimate the throughput of processing a work unit on each PPE. Note, the work unit size may vary on different PPEs, e.g., the CPU and the GPU may have different work unit sizes. In out settings, the work unit size varies among query-level, operator-level and OpenCL-kernel-level. Second, we obtain the current workload of each PPE (i.e., how much pending workload to finish). Third, we pick a PPE with the highest throughput for those PPEs whose current workload level is within a predefined threshold T_0 to the average workload among all PPEs. If that PPE cannot be found, we simply pick the one with the lowest workload. The on-line scheduling algorithm considers both hardware capability and current workload of each PPE. T_0 is a tuning parameter to adjust the two considerations mentioned above. In experiments, we choose T_0 =20% by default. On CPU-only or GPU-only architecture, the scheduling algorithm degrades to FIFO.

Cost model. In order to allow adapters to plug in architecture-aware cost estimations, the cost model offers two interfaces for adapters to instantiate for each primitive and operator. One interface is to count the number of memory blocks referenced by the PPE, and the other interface is to calculate the instruction execution time for the PPE. The later interface is simply an empty function, because the instruction cost is architecture dependent in OpenCL. The memory cost estimation is simply counting the number of memory blocks accessed in the query processing operation. By default, our cost model does not assume the existence of cache. We adopt very standard I/O model to estimate the cost [17].

Adapters. Our current implementation of adapters is simple, with the following major purposes. First, the adapter performs calibrations on the target architecture to obtain some important parameters. One important issue is whether the target architecture has a cache. If so, we need to measure the cache parameters such as cache line sizes and cache capacity. Second, the adapter performs architecture-aware tuning. On the CPU-only and the GPU-only architecture, work unit sizes are calibrated with the approach proposed in the previous studies [9, 3]. For CPU-GPU and APU architectures, we not only calibrate the suitable work unite sizes for both PPEs, but also calibrate the interconnect bandwidths among PPEs. Third, developer needs to override the interfaces defined in the cost model and the execution engine. Based on the cost functions, OmniDB can choose the most efficient primitives, access methods and operators for the target architecture.

Put it all together. We briefly present how OmniDB adapts to the four different architectures in the experiments.

Homogeneous architectures. We view the CPU-only or the GPU-only architecture as one PPE, because the data-parallel design of the execution engine can take advantage of the parallelism. Our scheduling algorithm degrades to FIFO. For the same query processing operations, the CPU mostly has a larger work unit size in numbers of tuples to process than the GPU, because of the more powerful CPU core design and larger cache.

Heterogeneous architectures. We view the CPU-GPU

and the APU architecture as two PPEs with different interconnect bandwidths, which equal the PCI-e bandwidth and the memory bandwidth, respectively. The throughput calculation in the scheduler takes that bandwidth information into account. Also, adapters may choose work unit definitions with different granularities: query-level, operator-level and OpenCL-kernel-level, where the scheduling decision is made per query, per operator and per OpenCL kernel, respectively. Query-level scheduling has the minimum data transfer between the CPU and the GPU, whereas OpenCL-kernel-level scheduling is the most fine-grained to exploit different capabilities of the CPU and the GPU.

There are other system components in our demo to help users understand the detailed performance behavior of OmniDB. We leverage vendor specific profiler from hardware vendors (e.g., Intel VTune, AMD CodeAnalyst Performance Analyzer and NVIDIA command line profiler).

4. DEMO PLAN

We briefly present the demo setup and objectives.

4.1 Demo Setup

We evaluate OmniDB on four target architectures, namely, CPU-only, GPU-only, (classic) CPU-GPU and AMD APU. We will use remote access during the demo. Due to space limitations, we present the detailed demo setup in Appendix A.

4.2 Demonstration Objectives

Portability. Portability is an important feature of OmniDB. We demonstrate system internals of OmniDB in two ways. First, we will make a poster with more detailed descriptions on the system internals, and also add one example to show the workflow of evaluating a query. Second, we will make OmniDB open-sourced, and will briefly go through our code base to the audience.

Efficiency. We shall demonstrate the efficiency of OmniDB in three aspects.

First, we shall evaluate the effectiveness of adapters on different architectures. As an example, we show the performance impact of different work unit sizes on homogeneous architectures: CPU-only and GPU-only. Overall, a suitable work unit size improves the query processing performance. Take hash joins as an example. The hash join with the suitable work unit size is up to 28%, 24% and 27% faster than that of other work unit sizes on the Intel CPU, AMD GPU and NVIDIA GPU, respectively.

Second, we assess the impact of the scheduling algorithm with different work unit definitions on heterogeneous architectures. The baseline scheduling algorithm is FIFO. Our scheduling algorithm achieves a higher throughput than the baseline algorithm, with the improvement of 8–33% and 4–19% on the CPU-GPU and the APU architectures, respectively. Among different work unit definitions on the specific architecture, OpenCL-kernel-level is the most efficient for APU, and query/operator level is the most efficient on the CPU-GPU architecture. This is mainly due to the different interconnects between the CPU and the GPU in those two architectures. A GUI is used to dynamically visualize the workloads in different PPEs.

Third, we demonstrate the profiler result. For example, from NVIDIA GPU profiler, we show partitioning reduces the L2 cache misses of hash joins by 20% on the NVIDIA GPU, in comparison with simple hash joins.

5. SUMMARY

OmniDB demonstrates that portability and efficiency of query processing on different parallel CPU/GPU architectures can be elegantly supported with a kernel-adapter approach. As the heterogeneity of parallel architectures, we believe that a portable and efficient query processor becomes more and more desirable. This demonstration presents our initial efforts in designing and implementing OmniDB. More work should be done along the direction. First, we plan to evaluate our system on other architectures such as Intel Xeon Phi. Second, we plan to evaluate OmniDB in comparison with existing architecture-aware query processors like GPUQP and StagedDB.

6. ACKNOWLEDGEMENT

The authors would like to thank anonymous reviewers for their valuable comments. This work is supported by a MoE AcRF Tier 2 grant (MOE2012-T2-2-067) in Singapore and an Inter-disciplinary Strategic Competitive Fund of Nanyang Technological University for "C3: Cloud-Assisted Green Computing at NTU Campus".

7. REFERENCES

[1] A. Ailamaki, N. K. Govindaraju, S. Harizopoulos, and D. Manocha. Query co-processing on commodity processors. In *VLDB (tutorial)*, 2006.

[2] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[3] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[4] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 2007.

[5] M. Daga, A. M. Aji, and W.-c. Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *SAAHPC*, pages 141–149, 2011.

[6] K. Fatahalian and M. Houston. A closer look at gpus. *Commun. ACM*, pages 50–57, 2008.

[7] S. Harizopoulos and A. Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[8] B. He, H. P. Huynh, and R. Mong. Gpgpu for real-time data analytics. In *IEEE ICPADS*, pages 945–946, 2012.

[9] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, pages 21:1–21:39, 2009.

[10] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. In *PVLDB*, pages 1–12, 2013.

[11] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *DaMoN*, pages 55–62, 2012.

[12] C. Kim and et al. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, pages 1378–1389, 2009.

[13] C. Kim and et al. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.

[14] M. M. Lehman. Programs, life cycles, and laws of software evolution. In *Proceedings of the IEEE*, pages 1060–1076, 1980.

[15] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, pages 108–124, 1996.

[16] M. Stonebraker and et al. C-store: a column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 3rd edition, 2007.

APPENDIX

A. MORE DEMO SETUP

Our demonstration is conducted on four different architectures on three machines (PCs A, B and C in Table 1).

- CPU-only: The CPU is Intel Xeon E5506 on PC A.
- GPU-only: The GPU is AMD FirePro V7800 on PC B, or NVIDIA Quadro 5000 on PC C.
- CPU-GPU: It uses the CPU-GPU architecture of PC A.
- APU: It uses the APU of PC B.

Table 1: Machine configuration. PCs A and C have discrete CPU-GPU architectures and PC B has a coupled CPU-GPU architecture.

	PC A	PC B	PC C
CPU	Intel Xeon E5506, 4 cores (2.13GHz), 4MB L2 cache	AMD A8-3870k, 4 cores (3.0GHz), 4MB L2 cache	Intel Xeon E5506, 4 cores (2.13GHz), 4MB L2 cache
GPU	ATI FirePro v7800, 1440 cores (700 MHz)	AMD A8-3870k (Radeon HD 6550D), 400 cores (600MHz)	NVIDIA Quadro 5000, 352 CUDA cores (1026MHz),
DRAM (GB)	8GB for CPU, 2GB for GPU	8GB shared by CPU and GPU, 2GB dedicated for GPU	8GB for CPU, 2.5GB for GPU

Table 2: Queries

ID	Name	Queries
Q1	SEL	SELECT R.a1 FROM R WHERE Lo $\leq$ R.a1 $\leq$ Hi
Q2	AGG	SELECT max(R.a1) FROM R
Q3	ORD	SELECT R.a2 FROM R WHERE Lo $\leq$ R.a1 $\leq$ Hi ORDER BY R.a1
Q4&Q5&Q6	EJ	SELECT R.a1 FROM R,S WHERE R.a1=S.a1

In our demonstration, we will submit custom queries to our synthetic data sets. Each of relations `R` and `S` consists of n four-byte integer attributes, a_1, a_2, ..., and a_n. The queries are shown in Table 2. All attributes are with random 32-bit values. To evaluate different join algorithms, we manually fix Q4, Q5 and Q6 to use indexed nested-loop joins, sort-merge joins and hash joins, respectively. In evaluating the efficiency of scheduling algorithms, we randomly select 30 instances of queries from Q1–Q6. The number of threads in submitting queries to be four so that the system is at high utilization. We also evaluate different data sizes (by default, both `R` and `S` have 8 millions of tuples each and $n = 2$), and evaluate different selectivity by varying Lo and Hi. We will also consider TPC-H queries.