Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware

Tim Gubner (2 November 2020)
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
tim.gubner@cwi.nl
Supervised by Peter Boncz

***Abstract*—Modern hardware tends to become increasingly heterogeneous which leads to major challenges for existing systems: (a) Given that performance on modern hardware should be maximized, hardware features should be fully exploited. Together with the increasing heterogeneity this leads to more complex systems. (b) In general it is non-trivial for a system to determine the most efficient way to execute a program on a specific piece of hardware. Based on (a) and (b) we believe it is necessary to extend code generation in data processing systems.**

First, to mitigate the increasing complexity we propose the usage of domain-specific languages (DSL) that abstract specific details away. Our DSL is based on data-parallel operations and control-flow statements which allows to easily exploit SIMD (on multiple architectures: CPU, GPU etc.). We briefly sketch our idea of such a DSL.

Second, we propose a virtual machine executing this DSL. We plan to exploit different implementation flavors (adaptivity) and dynamically compile & optimize hot paths in the program (JIT-compilation). We sketch ideas about which paths to compile, how to achieve adaptive execution of different JIT-compiled paths and elaborate a bit more on our ideas on workload-specific optimizations.

Finally, we present our plan for future research and ideas for major publications.

***Keywords*—heterogeneous hardware; domain-specific language; SIMD; data-parallelism; virtual machine; just-in-time compilation; adaptivity; dark silicon**

I. Introduction

In recent years modern hardware has become increasingly heterogeneous and future hardware will be even more heterogeneous. Today's CPUs include multiple (homogeneous) cores, specialized circuits for efficient en-/decryption, and sometimes complete GPUs or FPGAs. These components reside on the same chip and can be dynamically switched on when needed. This trend towards more heterogeneous integrated components that are only turned on for specific tasks is known as *dark silicon* [27].

In order to fully leverage modern heterogeneous hardware, and maximize performance, modern data processing systems face two types of challenges:

(a) hardware challenges. Systems become increasingly complex as underlying hardware diverges and more architectural details are exploited.

On a low-level, FGPAs, GPUs and CPUs require different programming models. Programming CPUs and GPUs might seem somewhat similar but with the increasing complexity of CPU instruction sets for specialized operations (e.g. MMX, SSE, SSE2, SSE3, SSE4.2, AVX, AVX2, AVX-512) exploiting them - typically - requires manual steps. All in all FPGAs, GPUs and CPUs share one aspect: To achieve peak-performance, potential parallelism has to be exploited.

From a high-level perspective, one would want to exploit hardware-specific architectural advantages: CPU-based systems may exploit the memory cache hierarchy, as in MonetDB/X100 [4], whereas GPU-based systems would leverage the high bandwidth to GPU's private memory, as in [7], [22] and [30]. FPGA-based systems would exploit the additional computational flexibility given through the ability to program circuits, as in [16], [25] and [5].

Previous approaches, such as VOODOO [23], Weld [19] and MRQL [8], proposed domain-specific languages (DSLs) that abstract implementation and hardware-specific details away. However, these attempts impose two challenges: First, the simplifying assumptions these make on data storage to make data fit across all platforms make it hard to leverage existing data ordering/clustering and, second, there is no generic way to exploit specialized instructions as this requires matching a code pattern to the instruction's specific use-case. We believe that to solve these challenges, it is necessary to design a different DSL.

(b) algorithmic challenges. Furthermore, even for CPUs, it is unclear how to generate code the fastest implementation for a query in an automated manner. State-of-the-art systems are HyPer [17] and MonetDB/X100 [4]. HyPer generates static LLVM code which is CPU-efficient but lacks the ability to fully take advantage of hardware parallelism such as single-instruction-multiple-data (SIMD) and parallel memory access. MonetDB/X100 relies on efficient interpretation by operating on cache-resident chunks of data aka *vectorized execution*. By encapsulating parallelism on operations of chunks, it provides a natural way to exploit SIMD and parallel memory access. This allows micro-adaptive optimizations [24], as decisions, to adapt to changed or - potentially - unexpected conditions, can be efficiently made between these operations.

As an example consider query 1 of the widely-used TPC-H benchmark: In [17] HyPer claims the fastest time whereas in [12] vectorized execution can beat a program similar to

HyPer's statically generated code by applying a mix of optimizations (i.e. smaller data types and an adaptively triggered pre-aggregation). These optimization can be generated in static code, although this would involve generating code for all variants with and without these optimizations potentially leading to code explosion. In practice, this might get even worse. Consider a modern in-memory database system, employing compression to fit more data into main memory and bypass bus bottlenecks. In case of compression it is (practically) impossible to always generate all possible combinations of compression methods, data types and operations on this data – as a vectorized engine would need. Note that this assumes the possibility of compression techniques within one column to change (e.g. block by block) in order to adapt compression methods to the data in each block and keep a good compression ratio.

Based on (a) and (b), we believe it is necessary to extend code generation in data processing systems. We propose the usage of a framework consisting of a DSL and a virtual machine (VM) executing the DSL.

The DSL hides implementation details while keeping data ordered between operations, and also serves as a intermediate representation between optimizations. it is based on a set of primitive operations that encapsulate (potentially) data-parallel operations. In order to further exploit specialized operations (on future hardware) new operations can be added to the language.

We propose to create a new kind of adaptive cross-platform VM that executes this DSL. Leveraging just-in-time (JIT) compilation, it adaptively mixes different execution strategies (interpretation, compilation and different compiled plans) whenever it turns out to be a viable choice. By using profiling information (performance counters, number of calls, number of tuples), the VM is able to observe and analyze the current workload and use the knowledge gained to specialize the program for the current workload. In case of compression in a database system, this means that the program may only contain the code of the current combination of compression techniques. If a different technique is encountered, the VM has to choose a different execution strategy (either generate different code, interpret, or a mix both).

II. Domain-specific language

The DSL's purpose is to be a layer between the higher layers, say, the *front-end* (query compiler, more user-friendly programming language for user-defined functions (UDFs) or application-code around the framework etc.), and the run-time/VM (hardware-specific code generation and execution).

This DSL employs data-parallel operations on arrays. This gives the VM the flexibility to execute the function in multiple ways (by simply rewriting the program). For example, one could process: Column-at-a-time, similar to MonetDB [14]; chunk-at-a-time, similar to MonetDB/X100 [4]; or tuple-at-a-time, similar to HyPer [17].

Unlike VOODOO [23], Weld [19] and MRQL [8] it includes control-flow operations. This allows the program to be implemented using specialized operations on chunked data (i.e. naturally vectorized), which can *easily* be transformed into (simpler) column-at-a-time or tuple-at-a-time execution strategies [1].

TABLE I
Data-parallel skeletons

Name	Purpose
map	Element-wise application of a function f on $\vec{v}$
filter	Element-wise selection using predicate p on $\vec{v}$
fold	Reduce $\vec{v}$ using initial value i and reduction function r
read	Consecutive read from position i in $\vec{d}$
write	Consecutive write $\vec{v}$ to location i of $\vec{d}$
gather	Read from locations $\vec{i}$ in $\vec{d}$
scatter	Write $\vec{v}$ to locations $\vec{i}$ of $\vec{d}$ using function f to handle conflicts
gen	Fill array using function f
condense	Eliminate selection vector from $\vec{v}$. Note that filters do not physically modify the flow, instead they calculate a selection vector
merge	Abstract merge for `MergeJoin`, `MergeDiff`, `MergeUnion` ...
...	

In addition to the mentioned transformation between execution strategies, the DSL allows an easy implementation of:

- Deforestation to eliminate intermediate data structures [28], by fusing operations (essentially loop fusion on the data-parallel operations) together.
- Pipeline-building, as each pipeline begins with a new loop that first builds chunks of data.
- Parallelization, through the manipulation of loop boundaries. However, morsel-driven parallelism [15] requires the language to support dynamic loop boundaries.

Note that this is only a subset of the possible transformations, e.g. types can be refined based on statistics (compact data types [12]) or data layout can be modified (NSM vs. DSM [33]).

The envisioned language itself revolves around data-parallel patterns/skeletons ([6], essentially higher order functions). These skeletons operate on arrays of data (both with a fixed length). Scalar values can be seen as arrays with length 1. A basic set of these skeletons can be seen in Table I. Note that the DSL can be extended by adding new skeletons.

In order to represent relational queries, the language has been extended by control flow and mutable variables. The astute reader might wonder why this approach has been chosen instead of nice and simple constructs such as Monad Comprehensions [11], as e.g. used in Weld [19]. The reason for the additional complexity is that matching per-tuple expressions to efficient and specialized data-parallel operations is non-trivial. Also these approaches do not allow some efficient algorithms to be implemented, such as `MergeJoin`s.

Apart from skeletons, the DSL implements expressions (constants, function application, variables), control-flow (infinite loop, break, if-then-else) and state maintenance (define &

[1] First one has to manipulate the array lengths, followed by partial evaluation [9] which will remove the loop implementing the chunking.

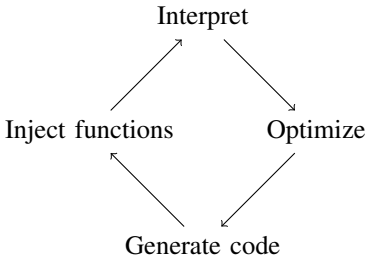

Fig. 1. VM state machine

update a mutable variable), as well as, assignments to variables (to allow sharing intermediate results) and function definitions.

III. Virtual machine

The VM described in the following executes the DSL described in Section II. The VM should incorporate a JIT compiler that uses feedback from runtime (profiling data) to optimize/specialize the program for the current workload, while trying to interpret cold code and short-running programs. Figure 1 shows the VM's state machine. Program execution starts with interpretation, meanwhile the VM collects profiling information (time spent in each operation, number of calls) to identify hot paths and potential targets for further optimization. At some point, the interpreter decides to optimize and will eventually generate optimized code which will get injected into the interpreter. Afterwards program interpretation continues with a partially optimized program.

A. Efficient interpretation

Interpreting the DSL leverages chunk-at-a-time efficient/vectorized interpretation, similar to MonetDB/X100 [4]. Hence, specialized functions that operate on a chunk of data in a tight loop are needed. We can generate and compile these functions during startup through our compilation infrastructure, such that they will be available during runtime with near to zero compilation effort.

During interpretation, a program (implemented in the DSL) will first be rewritten into a program using vectorized execution. Note that functions might still include complex operations (e.g. a `map` applying $f(a,b) = \sqrt{a^2 + b^2}$ to each item). These functions have to be normalized, which means, breaking them into simpler operations. In our example we split the function into: $f_1(a) = a^2 = f_2(b) = b^2$, $f_3(x,y) = x + y$ and $f_4(x) = \sqrt{x}$. For each of these operations the pre-compiled functions can be looked up and be called during runtime.

B. (Partial) Compilation

In order to improve performance of hot paths (frequently called expressions or long running operations), we compile program fragments into efficient machine-code.

The purpose of our partial compilation is to minimize compilation effort (optimizer passes tend to take longer with an increasing amount of code) while focusing on the hot paths in the program. We do this by generating new functions which compute parts of the program [2]. Each new function can be compiled and optimized separately and directly plugged into the interpreter.

```
mut i
mut k
i := 0
k := 0
loop
  let input = read i some_data in
    let a = map (\x -> 2*x) input in
      let t = filter (\x -> x>0) a in
        let b = condense t
          write v i a
          write w k b
          i := i + len(a)
          k := k + len(b)

  if i >= 4096 then
    break
```

Fig. 2. Example in our DSL. Program that reads `some_data` (array of integers) and outputs (a) twice the value of each integer and (b) that number, only if bigger than zero. Both are written consecutively into output arrays `v` and `w`.

Consider a program in our DSL such as the one in Figure 2. This example multiplies the first 4096 numbers from `some_data` by two, writes it to `v` afterwards and writes to `w` all the elements that are bigger than zero. It covers important parts of the in Section II envisioned DSL: It defines the mutable variables `i` and `k`. These variables will then be initialized to `0` using the `:=` operator which, as a side effect, assigns the result of the expression (here `0`) to the mutable variable. This is followed by the `loop` statement which executes its subprogram in an infinite loop. In this case its subprogram consists of two statements: The first processes the input data. The latter checks whether $i \geq 4096$ and in that case terminates the loop (`break`). Focusing on the subprogram processing the input, one can observe cascaded bindings of expressions to, in each iteration of the loop, immutable variables (`let A = B in subprogram`). Each binding makes use of data-parallel skeletons (`read`, `map`, `filter` and `condense`). In case of `map` and `filter` lambda functions are used to transform the input and to compute the predicate, respectively. In the subprogram of the last binding stateful statements can be found that either write expressions to the output (`write` skeleton) or assign new values to mutable variables `i` and `k`.

Figure 3 shows the partial dependency graph of the example in Figure 2. As can be seen, an iteration of the loop has been split into two functions: One multiplying the input by two, after reading the input and the other one filters the computed intermediate.

In order to find such functions in an automated manner, we propose to greedily partition the dependency graph. Starting with an initially empty set of functions R, we go over the graph and select the most expensive node (operation). From this node we greedily add neighbor nodes until one of our

[2]Note that these functions do not necessarily partition the program, as (a) they might overlap and (b) they do not necessarily cover the whole program.

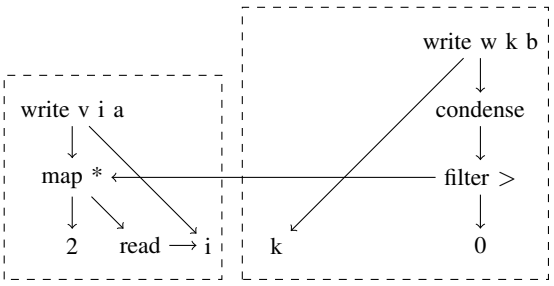

Fig. 3. Dependency graph for an iteration of `loop` in Figure 2 (excluding updating mutable variables and control-flow) partitioned into two compilable functions.

heuristic constraints (described later) is violated. While we are doing this we mark these nodes as visited. All newly marked nodes belong to one function f and we add f to R ($R' := R \cup \{f\}$). Afterwards, we go to the next expensive (unvisited) node and do the same.

This ends when either a threshold is reached or no nodes can be visited. The remaining nodes can either be compiled or interpreted.

Furthermore we propose that these heuristic constraints should be used:

- We do not allow more than n inputs/intermediates per function whereas n depends on the size of the Translation look-aside buffer (TLB). This prevents TLB thrashing in the generated functions.
- Furthermore we do not allow to include some operations inside functions, such as `filters`. This restricts the impact of branch miss-predictions and data dependencies caused by the computation of the selection vector. Also, more complex operations, such as non-trivial string operations (string length, string concatenation etc.) should not be included, as they hinder vectorization.

Conceptually this replaces operations with optimized JIT-ed operations whereas "replacing" means that the VM adds a new possible execution path (trace). The repetition of this algorithm will eventually lead to many of these traces, each optimized for a specific situation. The VM then chooses - based on the current situation - a trace, if it already learned about that situation, or falls back to interpretation.

C. Workload-specific optimization - Beyond micro-adaptivity

One of the advantages of (classical [3]) JIT compilation, as in GraalVM [29], SPUR [2], Hotspot [18], Hotpath [10], PyPy [3] and luajit [20], is the potential to exploit workload-specific optimizations. This has been partially exploited in databases through micro-specialization by [31] and [13]. We aim to generalize these techniques and apply them in a much bigger scope: Once the workload changes (triggered by program itself or by profiling information), we can either interpret the program or optimize for the new workload. Note that this allows for many techniques to be implemented.

For example, assuming the input columns are compressed, we can leverage the fact that the compression techniques stay the same in a chunk of data, say a block, and exploit *compressed execution* [1]. When iterating to the next block the compression techniques might change. Once the VM notices that change, it will fall back to decompression (as, e.g. in [32]) and interpretation. Later, it can provide a (partially) compiled and optimized alternative.

Additionally, one could also *specialize for different selectivities* i.e. when the VM notices that no (resp. many) tuples are filtered out, it can compile code to fully (resp. selectively) evaluate expressions. In the (close to) non-selective case one could implement selective processing using bitmap and build longer functions by potentially inlining the `filter`. This would allow optimizations to be done on a larger scope and might allow full SIMD-ization of the function. In the other extreme one could use selection vectors and restrict the `filter` to one function.

Another possibility is the *on-the-fly reordering of selective operators*. Consider a chain of two `HashJoin` operators A and B. We could filter the tuples using the A first and later B (essentially executing the `SemiJoin` first), when A eliminates more tuples from the flow. During runtime the order of these operations could change dynamically based on the observed selectivity.

IV. Plan

Based on our idea to exploit dynamic JIT compilation in programs/queries, we propose the following research plan, that consists of three steps, each envisioned to take one year. Each step should lead to a major publication.

We start with developing the basic framework (interpreter and code generator) which we plan to integrate into an open-source system such as Peloton [21], PostgreSQL [26] or MonetDB [14]. Our first target is to create this framework on CPUs only, and implement various different execution skeletons for it. Specifically, we would like the same system to be able to either use vectorized execution, or tuple-at-a-time JIT compilation, as such mimicking the MonetDB/X100 and HyPer approaches inside the same framework. Concretely, this target entails significant software and experimental work, but also a paper that motivates the DSL, and describes its syntax and semantics formally.

The second target is to make the VM dynamic. Thus, it should observe and and analyze the current workload, make JIT compilation decisions and choose between multiple execution strategies. These different strategies could relate to selectivity of filters in select/project pipelines (e.g. sometimes delaying filtering in favor of SIMD projection execution), the applicability of Bloom-filters in selective hash-joins, the re-ordering of join operations inside the same pipeline, the detection of opportunities to execute expressions in smaller data types, and compressed execution opportunities in general.

[3] "Classical" because in the database context JIT compilation tends to be confused with ahead-of-time compilation where code for a query is generated, e.g. in case of HyPer [17]

A third target is to support multiple hardware platforms in the VM, this FPGA or GPU, but most prominently, GPUs. Please note that we do not envision GPUs to implement the full DSL, necessarily, we might concentrate their use around certain operations where their capabilities best come to light, This step addresses the hardware challenge (b), and will again involve significant software work. In this integration, we will show that the VM can execute a query on multiple hardware platforms, making adaptive decisions which strategy to use (previous step) but also on which hardware.

REFERENCES

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM, 2006.

[2] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: A trace-based jit compiler for cil. *SIGPLAN Not.*, 45(10):708–725, Oct. 2010.

[3] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[4] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[5] Y.-T. Chen, J. Cong, Z. Fang, J. Lei, and P. Wei. When apache spark meets fpgas: A case study for next-generation dna sequencing acceleration. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 64–70, Berkeley, CA, USA, 2016. USENIX Association.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[7] B. Falsafi, B. Dally, D. Singh, D. Chiou, J. J. Yi, and R. Sendag. Fpgas versus gpus in data centers. *IEEE Micro*, 37(1):60–72, Jan. 2017.

[8] L. Fegaras. An algebra for distributed big data analytics. 2016.

[9] Y. Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, Dec 1999.

[10] A. Gal, C. W. Probst, and M. Franz. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, VEE '06, pages 144–153, New York, NY, USA, 2006. ACM.

[11] T. Grust. *Monad Comprehensions: A Versatile Representation for Queries*, pages 288–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[12] T. Gubner and P. Boncz. Exploring query execution strategies for jit, vectorization and simd. In *ADMS 2017*, 2017.

[13] C.-P. Haensch, T. Kissinger, D. Habich, and W. Lehner. Plan operator specialization using reflective compiler techniques. In *BTW*, pages 363–382, 2015.

[14] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, page 2012.

[15] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[16] R. Mueller, J. Teubner, and G. Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[17] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[18] M. Paleczny, C. Vick, and C. Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[19] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics.

[20] M. Pall. Luajit 2.0 intellectual property disclosure and research opportunities. http://lua-users.org/lists/lua-l/2009-11/msg00089.html. Accessed: 2017-02-02.

[21] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. Mowry, M. Perron, I. Quah, S. Santurkar, A. Tomasic, S. Toor, D. V. Aken, Z. Wang, Y. Wu, R. Xian, and T. Zhang. Self-driving database management systems. In *CIDR 2017, Conference on Innovative Data Systems Research*, 2017.

[22] H. Pirk, S. Manegold, and M. Kersten. Waste not ... efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 508–519, March 2014.

[23] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, Oct. 2016.

[24] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1231–1242, New York, NY, USA, 2013. ACM.

[25] D. Sidler, Z. Istvan, M. Owaida, K. Kara, and G. Alonso. doppiodb: A hardware accelerated database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1659–1662, New York, NY, USA, 2017. ACM.

[26] M. Stonebraker and L. A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 340–355, New York, NY, USA, 1986. ACM.

[27] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *In Proceedings of the 49th Design Automation Conference*, 2012.

[28] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *European Symposium on Programming*, pages 344–358. Springer, 1988.

[29] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204. ACM, 2013.

[30] Y. Yuan, M. F. Salmi, Y. Huai, K. Wang, R. Lee, and X. Zhang. Spark-gpu: An accelerated in-memory data processing engine on clusters. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 273–283, Dec 2016.

[31] R. Zhang, S. Debray, and R. T. Snodgrass. Micro-specialization: Dynamic code specialization of database management systems. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 63–73, New York, NY, USA, 2012. ACM.

[32] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.

[33] M. Zukowski, N. Nes, and P. Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, pages 47–54, New York, NY, USA, 2008. ACM.