Vectorized Data Processing on the Cell Broadband Engine

Sándor Héman Niels Nes Marcin Zukowski Peter Boncz

CWI, Kruislaan 413
Amsterdam, The Netherlands
{Firstname.Lastname}@cwi.nl

ABSTRACT

In this work, we research the suitability of the Cell Broadband Engine for database processing. We start by outlining the main architectural features of Cell and use micro-benchmarks to characterize the latency and throughput of its memory infrastructure. Then, we discuss the challenges of porting RDBMS software to Cell: *(i)* all computations need to SIMD-ized, *(ii)* all performance-critical branches need to be eliminated, *(iii)* a very small and hard limit on program code size should be respected.

While we argue that conventional database implementations, i.e. row-stores with Volcano-style tuple pipelining, are a hard fit to Cell, it turns out that the three challenges are quite easily met in databases that use column-wise processing. We managed to implement a proof-of-concept port of the *vectorized query processing* model of MonetDB/X100 on Cell by running the operator pipeline on the PowerPC, but having it execute the vectorized primitives (data parallel) on its SPE cores. A performance evaluation on TPC-H Q1 shows that vectorized query processing on Cell can beat conventional PowerPC and Itanium2 CPUs by a factor 20.

1. INTRODUCTION

The Cell Broadband Engine [9] is a new heterogeneous multi-core CPU architecture that combines a traditional PowerPC core with multiple mini-cores (SPEs), that have limited but SIMD- and stream-optimized functionality. Cell is produced in volume for the Sony Playstation3, and is also sold in blades by IBM for high-performance computation applications (we used both incarnations). The Playstation3 Cell runs at 3.2GHz and offers 6 SPEs providing a computational power of 6x25.6=154GFLOPs, which compares favorably to "classical" contemporary CPUs, which provide up to 10GFLOPs.

In this paper, we research the suitability of the Cell Broadband Engine for database processing. This is especially interesting for highly compute-intensive analysis applications, like data warehousing, OLAP and data mining. Therefore, we use the TPC-H data warehousing benchmark to evaluate the efficiency of running database software on Cell.

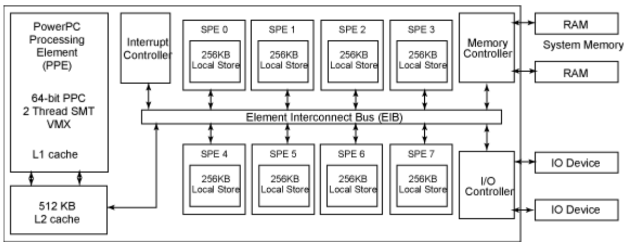


Figure 1: Cell Broadband Engine Architecture

There turned out to be three main challenges in porting RDBMS software to Cell:

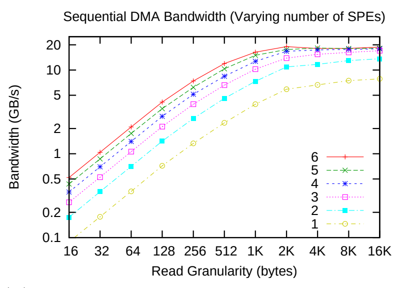
(i) all computations need to SIMD-ized, as the SPEs support SIMD instructions **only**. While there has been work on using SIMD in database systems [15], this work needs to be extended to enable full database operation on Cell. We address this need partly in Section 4, by contributing a new method to process grouped aggregates (i.e. `SELECT .. GROUP BY`) using SIMD instructions.

(ii) all performance-critical if-then-else branches need to be eliminated, as SPEs combine a high branch penalty with a lack of branch prediction. Some database processing techniques such as buffered execution of relational operators [16], predicated selection [13] but also vectorized execution [3], can be used to reduce the impact of these branch misses.

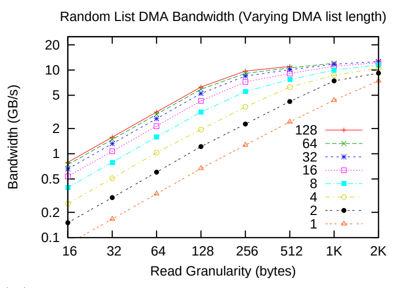
(iii) there is a very small yet hard limit on program code size, as in each SPE data *plus code* should not exceed 256KB. It turned out impossible to run "conventional" database engines – such as Postgres – on the SPEs, as their code measures MBs rather than KBs. The code size challenge implies that on Cell, a database system must not only manage the data cache, but also its own instruction cache!

We report here on our initial experiences of porting the vectorized query processing model of MonetDB/X100 [3] to Cell. This port uses the PowerPC to run the relational operator pipeline, but executes its data-intensive *vectorized primitives* data parallel on the SPEs, using a small run-time system that manages transfer of data and instructions. At the time of this writing, the port only consists of this run-time system, together with the operators and primitives needed by TPC-H Query 1.

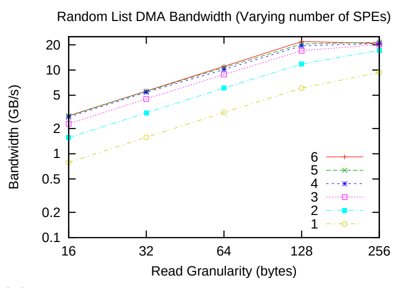
Outline & Contributions. Section 2 summarizes the Cell architecture, and characterizes its programmable DMA memory infrastructure using micro-benchmarks. In Section 3 we discuss how various DBMS software architectures



(a) DMA read bandwidth as a function of bytes per transfer, for varying number of SPEs



(b) List-DMA read bandwidth as a function of bytes per transfer element, for varying DMA-list lengths (1 SPE)



(c) List-DMA read bandwidth as a function of bytes per transfer element, for varying number of SPEs (list length 128)

Figure 2: DMA read bandwidth Micro-Benchmarks (logarithmic scale)

could be mapped to Cell hardware. We cover three main processing models: classical Volcano-style NSM tuple pipelining, column-wise materialization (MonetDB) and vectorized query execution (MonetDB/X100). Section 4 shows how various vectorized relational database operators can be implemented using SIMD instructions. Experiments with TPC-H Query 1 show that the vectorized query processing used in MonetDB/X100 can be a factor 20 faster on Cell than on contemporary CPU architectures. Wrapping up, we have related work in Section 5 and conclude in Section 6.

2. CELL ARCHITECTURE

Figure 1 shows a diagram of the Cell architecture. To the left is the PowerPC Processor Element (PPE), which is a general-purpose CPU, good at executing control-intensive code such as operating systems and application logic. The remaining eight cores are equivalent Synergistic Processing Elements (SPEs). The SPEs are optimized for compute-intensive tasks, and operate independently from the PPE. However, they do depend on the PPE to run an operating system, and in most cases the main thread of an application. The SPEs and PPE are connected using a 128-byte Element Interconnect Bus (EIB) that is connected to a 2-channel memory controller, with each channel being able to deliver 12.6GB/s of data, resulting in a theoretical maximum memory bandwidth of 25.2 GB/s.

Our main experimentation platform, a Sony Playstation 3 (PS3) game console, differs slightly from this architecture in that it has two of the eight SPEs disabled. The PS3 contains a Cell processor running at 3.2GHz, 256MB RAM, and runs the Linux operating system.

The SPE is an independent processor that runs threads spawned by the PPE. It consist of a processing core, the Synergistic Processing Unit (SPU), a Memory Flow Controller (MFC), and a 256KB local storage memory area (LS), that must keep both data and code. There is no instruction cache, which implies that code **must** fit in the LS. Although SPEs share the effective address (EA) space of the PPE, they cannot access main memory directly. All data an SPE wishes to operate on, needs to be explicitly loaded into the LS by means of DMA transfers. Once the data is in LS, the SPU can use it by explicitly loading it into one of its 128 128-bit registers. The SPE instruction set differs from the PowerPC instruction set and consists of 128-bit SIMD instructions **only**, of which it can execute two per clock cycle. The SPEs are designed for high-frequency (with a pipeline depth of 18), but lack branch prediction logic. While it is possible to provide explicit branch hints, if this is not done a branch costs 20 cycles, which implies that they should be avoided in performance-critical code paths.

Summing up, the SPE architecture requires careful engineering by the programmer to ensure that efficient branch-free SIMD code is generated, and to use parallel algorithms to exploit all SPEs, while keeping a careful eye on code size or even employing some dynamic code management scheme (see Section 2.2).

DMA Engine. An interesting aspect of SPE programming is the DMA Engine it exposes. The explicit memory access programming it enforces poses some extra work for the application developer, compared to normal cache-based memory access. Explicit memory access can, however, be an advantage for database software, as it provides full control of data placement and transfer, such that advance knowledge of data access patterns can be exploited. Previous work on data management using cache-less architectures has demonstrated that this is quite workable [6, 4] The DMA engine allows to request multiple (at most 256) memory blocks in one go ("List-DMA"). The practical minimal transfer unit (and alignment unit) is 128 bytes, while the maximum is 16KB. Each DMA transfer moves data between main memory and LS in asynchronous fashion, supporting both a polling and signaling programming model.

2.1 Memory Micro-Benchmarks

Potentially, the feature of List-DMA allows for efficient *scatter-gather* algorithms that gather input data from a large amount of random memory locations or scatter data output over a series of random locations. On normal cached memory architectures, such algorithms perform badly if the randomly accessed memory range exceeds the cache size, even if all cache lines are used fully. The reason is that optimum memory bandwidth is only achieved when sequential access triggers built-in hardware memory prefetching (e.g. a memory latency of 100ns and a cache line size of 64 bytes produces 640MB/s of random throughput, while sequential bandwidth on modern PCs gets to 4GB/s with prefetching). An example of a gather algorithm is Hash Join. For such algorithms, it is currently beneficial to perform additional partitioning steps (a scatter operation) to make the randomly accessed range fit the CPU cache first [10].

We use micro-benchmarks to investigate whether the Cell DMA Engine offers alternative ways of expressing data-intensive algorithms (e.g. is cache-, or rather LS-partitioning required at all for hash-based algorithms?).

Sequential Access. We conducted a micro-benchmark where we iteratively transfer a large region of main memory into the LS, in consecutive DMA transfers of x bytes, for varying x. Figure 2(a), shows that DMA latency dominates using small transfer sizes, but good bandwidth is achieved with memory blocks $\geq$ 2KB, giving one SPE a maximum of 6GB/s (these sequential accesses use only a single memory channel). If multiple SPEs perform the same micro-benchmark, we observe that already around 1KB transfers the SPEs fight for bandwidth. When all SPEs demand large sequential blocks simultaneously, we achieve a total score of 20GB/s memory read bandwidth, thus relatively close to the theoretical maximum of 25.6GB/s.

List-DMA allows to instantiate a list of *(size, effective_address)* pairs in LS to pass to the MFC for processing in one go. Figure 2(b) shows the single-SPE bandwidth, as a function of the transfer size per list element, where each list element reads from a random, 128-bit aligned location. The linear bandwidth increase up to 128 byte transfers is simply caused by the fact that all data transfers over the EIB have a minimum 128-byte granularity. For transfer sizes below 128 bytes, one is simply wasting bandwidth. We also see that beyond list-DMA bandwidth continues to improve with larger transfer sizes and approaches 10GB/s, surpassing the 6GB/s achieved with sequential access. The reason is that these random transfers use both memory channels. The figure furthermore shows that increasing the DMA list length keeps improving performance.

Figure 2(c) shows that when more than one SPEs are performing scatter/gather DMA at the same time, the bus again gets saturated, achieving peak 22GB/s memory bandwidth already at 128-byte transfer sizes.

The fact that random List-DMA is able to achieve high bandwidth indicates that Cell algorithms may indeed forgo LS-partitioning and work in a scatter/gather fashion directly on RAM. However, three caveats apply. First, it is essential that algorithms use a 128-byte granularity for memory access, thus making full use of the EIB "cache lines" to avoid bandwidth waste. Secondly, as SPEs should operate in parallel, the per-SPE usable throughput is limited to roughly 3GB/s (i.e. 1 byte per cycle). Given the 2-per-cycle throughput of the SIMD instructions that may process two 16 byte inputs, there is a distinct danger of becoming LS bandwidth-bound (16-bytes/cycle max). Finally, as the LS is not a coherent cache, RAM-based scatter/gather algorithms must explicitly prevent that the same memory locations are updated multiple times by the same DMA-List command, which can significantly increase their complexity.

2.2 Code Management

As the SPE local storage (LS) is limited to 256KB and needs to be shared between both code and data, major software products such as database systems simply will not fit. The Octopiler research compiler, being developed by IBM [5], tries to hide code size limitations by automatically partitioning code into small enough chunks. It embeds in each SPE program a small runtime system called the *partition manager*. The compiler also translates calls to functions outside the current partition into calls into the partition manager. At runtime, when called, the partition manager brings in the desired partition – swapping out the current one – using a DMA data transfer, and then calls into the newly loaded function. Additionally, this compiler promises a 32KB software cache, that allows to transparently access RAM resident arrays with a 12 instruction latency, and many other advanced features. Regrettably, however, all these features are not yet available in the IBM compilers currently distributed. Thus, code partitioning on Cell remains a programmer responsibility, so we discuss various ways to do this.

Separate Binaries. Each SPE can only run one single-threaded program at a given time, and no operating system code is running in between. Such a program is spawned by the PPE as an SPE thread, which transfers the SPE binary to any number of SPEs and runs it till completion. The simplest approach to partitioning thus is to explicitly compile the program into separate binaries and let these be spawned as SPE threads by the PPE whenever appropriate. Regrettably, this approach is slow, taking approximately 3804000 cycles (1.2ms) on average per SPE, so it is only viable when one partition will run for at least 100 milliseconds (in which 300MB of data should be processed). A second disadvantage is that it is static: the partitions need to be defined at system compile-time, making it hard to e.g. adapt to the needs of a run-time query plan.

Overlays. While the IBM compiler does not yet deliver the fancy features it promised, it does allow for *code overlays*: small libraries of SPE code that are compiled into the main binary, but do not get loaded into the SPE upon thread creation. Only when a function from a certain overlay is called, the overlay is loaded into the SPE at run-time, and the function gets executed. At roughly 775 cycles, this approach is much faster then separate binaries, but is still static (and even if the overlay is already loaded, the function call overhead is still a hefty 236 cycles). Also, overlay technology depends on inter-procedural analysis and thus cannot deal with late binding and function de-referencing, often used to implement DBMS execution engines.

Manual Loading. It is perfectly possible for an SPE program to issue a DMA memory transfer and upon completion call into that location. This makes it a viable strategy for a database system to add code management to the list of database (optimization) tasks. Like the Octopiler partition manager, each SPE could run a small runtime system that waits for code and data requests from the PPE. When a request comes in, it loads the data and code (if not already present in LS) and executes the required operation. The overhead of this approach is similar to the overlay approach. A limitation of this approach is that the code snippets that get transferred need to be stand-alone functions (i.e. functions that do not rely on relocation and/or call other functions). The advantage of manual loading is that, unlike overlays, it is able to deal with late binding – as said an important feature for porting database systems. In our database experiments of Cell, we therefore have used the Manual Loading approach to code management.

3. DBMS ARCHITECTURE ON CELL

3.1 Classical NSM Tuple Pipelining

Conventional relational database architecture uses a disk-based storage manager using an NSM layout. Query execution uses a Volcano-style [7] iterator class hierarchy, where all relational operators (Scan, Select, Join, Aggregation) are instantiated as objects that implement an *open()*, *next()*, *close()* method interface. A full query plan is a tree of such objects, and the result is generated by calling *next()* on the root of the tree, which pulls data up by calling *next()* in its children (and so on), finally producing a single result tuple. This is then repeated until no more tuples are returned.

Conventional relational database systems have a large code base that does not fit the 256KB LS, and therefore we think it is an absolute necessity that Cell compilers support automatic code partitioning (see Section 2.2). As this is not yet the case, we did not really attempt to port a real system.

Even if code partitioning would be available, the cost of crossing code partitions could be a problem. It may be possible to overcome this problem by inserting Buffer() operators [17] in the query plan, that force the *next()* method of its child operator to be called many times, buffering the results, before passing them up higher in the pipeline.

Finally, Cell may be an interesting platform for compiled query execution, where a query plan generator emits (C/C++) program code that is compiled Just-in-Time (JIT). It has been shown that query-specific code generation can be more efficient than a query interpreter [12, 3]. On Cell, an additional advantage is that the generated binary is much smaller than the full DBMS, and likely fits in LS.

As the query interpreter is an important cause of the high amount of if-then-else branches in database code, JIT compilation likely reduces the effect of costly branches on Cell. Data related branches can be addressed by using *predicated* programming techniques [13]. The SIMD instruction set of Cell also provides explicit intrinsics for predication.

Finally, it is known that a wide range of database operations can be accelerated with SIMD instructions [15]. As SIMD instructions apply an operation on X consecutive values from the same column, as a preparation X NSM horizontal records need to be *packed* vertically to put them into a SIMD register. It has been shown that on-the-fly conversion of horizontal (NSM) layout into small vertical arrays can improve performance [11]. However, the memory-intensive work of navigating through an NSM disk block following data layout offsets to gather column data is not a strength of the SPEs. As mentioned, a vertical (DSM) storage scheme, as used in MonetDB stores the data in SIMD-friendly vertical layout upfront, avoiding the need of packing. Note that packing can also be avoided by using column-wise storage only within a disk block (i.e. PAX [1]).

3.2 MonetDB: Column Materialization

MonetDB is an open-source DBMS using vertically fragmented storage (DSM) supporting both SQL and XQuery.[1] Internally, it implements a physical column-algebra [2] using a column-wise materialization strategy, which means that each operator reads one or more input columns, represented as contiguous arrays in RAM, and stores back the output column in RAM. MonetDB can be ported to Cell by running its algebra interpreter on the PPE, and have it execute the column operations (data parallel) on the SPEs.

Arguably, column-wise processing overcomes all three main Cell efficiency challenges (full SIMD-ization, branch elimination, instruction cache management): *(i)* Column algebras carry out *one* basic action on *all* values of a column sequentially, and thus have a high instruction locality. This makes it easy to do explicit SPE instruction cache management. *(ii)* Column-wise storage yields array-loop intensive code patterns that often can be compiled into SIMD instructions automatically. *(iii)* Finally, column-wise execution lessens the performance impact of branches caused by the query algebra interpreter, as interpretation decisions are made for whole columns, rather than tuples.

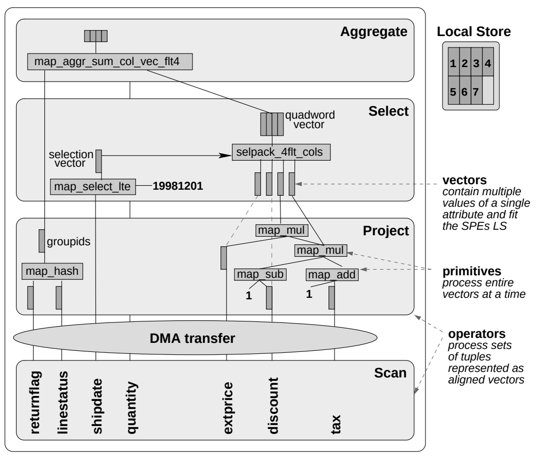


Figure 3: MonetDB/X100 on Cell

The full materialization strategy of MonetDB causes problems, however, if queries produce substantial intermediate results. In the case of our example query TPC-H Q1, this indeed occurs as the query starts with a selection that keeps 95% of the tuples. In case of Cell, we will see in Section 4.1 that this causes the SPEs to generate huge DMA traffic, such that performance becomes bus limited.

3.3 MonetDB/X100: Vectorized Processing

Contrary to MonetDB, the MonetDB/X100 system [3] allows for Volcano-style pipelining (avoiding materialization of intermediates). For disk storage, it can use both horizontal (PAX) and vertical (DSM) storage. Figure 3 shows an operator tree, being evaluated within MonetDB/X100 in a pipelined fashion, using the traditional *open()*, *next()*, *close()* interface. However, each *next()* call within MonetDB/X100 does not return a single tuple, as is the case in most conventional DBMSs, but a collection of *vectors*, with each vector containing a small horizontal slice of a single column. Vectorization of the iterator pipeline allows MonetDB/X100 *primitives*, which are responsible for computing core functionality such as addition and multiplication, to be implemented as simple loops over vectors. This results in function call overheads being amortized over a full vector of values instead of a single tuple, and allows compilers to produce data-parallel code that can be executed efficiently on modern CPUs. The vector size is configurable (typically 100-1000) and should be tuned such that all vectors needed for a query fit in the CPU cache (or LS – for Cell).

Cell Port. The *vectorized query processing* model of MonetDB/X100 can be mapped on Cell by running the relational operator pipeline (Scan,Select,Aggregation, etc) on the PPE. When a *next()* method needs to compute primitives, it sends a *primitive request* to all SPEs (data parallel on the vectors). *Double buffering* can be applied by executing a request only when a subsequent request is issued, using

[1]See `monetdb.cwi.nl`

the time in between to initiate DMA for code and data (if needed). As for code loading, we applied the manual loading approach described in Section 2.2 to load MonetDB/X100 vectorized primitives on demand to the SPEs.

In vectorized query processing, rather than writing all intermediate results to RAM, the result vectors coming out of vectorized primitives are kept in the local SPE memories, for use as input to the next operation. The main database system running on the PPE, allocates *vector registers* to primitive function inputs and outputs found in the query plan. These vector registers are symbolic representations of memory areas in the LS. Upon plan generation, the total number of vectors and their types are known, so a suitable vector size and *vector register allocation* can be chosen.

The Cell port of MonetDB/X100 is currently in a proof-of-concept stage. For the experiments presented in the next section, we hand-coded the PPE query plan and register allocation, and ported only the MonetDB/X100 primitives we needed to the SPE. Also, we re-used this code for vectorized query processing to emulate full column materialization (MonetDB), using an alternative plan that writes out each primitive output into a RAM-resident result column.

4. VECTORIZED SIMD PROCESSING

Projection. To integrate SIMD processing into a database kernel, it is advisable to let the compiler do as much of the work as possible. Implementing database operators as branch- and dependency free loops is crucial to make that possible. In MonetDB/X100, this holds automatically for most projection-related primitives, which are simple loops over vectors of the form:

```
for (i=0; i<n; i++)
   res[i] = input1[i] OP input2[i] ;
```

If we take for example addition on two floating point vectors, this will get compiled as if the code was explicitly SIMD-ized as follows (trailing tuples left out):

```
vector float *input1, *input2, *res ;
for (i=0; i < (n/4); i++)
   res[i] = spu_add(input1[i], input2[i]) ;
```

This adds four pairs of floats in parallel, obtaining a throughput of 1 tuple per SPE cycle.

Selection. To be able to exploit SIMD, the input vector arrays need to be aligned and organized sequentially in memory, so that multiple data items can be loaded into a SIMD register. The constraint of sequential input data, however, is violated by the way X100 originally implements selections, which it does by passing an optional *selection vector* to primitives, which is an integer array, containing the offsets of those tuples within the vector that are within the selection. This gives primitives of the form:

```
for (j=0; j < n; j++) {
    int i = sel[j] ;
    res[i] = input1[i] OP input2[i] ;
}
```

which decreases Cell throughput by a factor 20 (in case of 100% selection). To avoid this, instead of a positional selection vector we decided to use bit-mask selection vectors on Cell, which are aligned to data vectors, and contain bit-masks of either zeros or ones for non-selected and selected

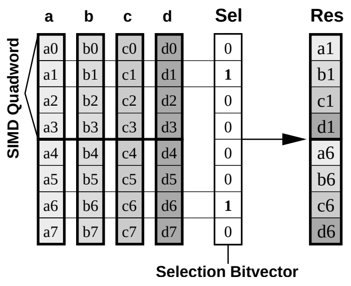


Figure 4: selpack SOA into AOS with selection

tuples respectively. This has the advantage that code remains SIMD-izable, as non-selected tuples can be quickly masked out whenever needed. A disadvantage is that non-selected tuples are still being processed and occupy space in the LS. Thus, if the selectivity is high, it may be better to compact the vectors, making them densely populated again.

SOA vs AOS. The representation used so far of horizontally aligned vectors, is called Structure of Arrays (SOA). SIMD operations can also be applied using an Array of Structures (AOS) layout. These SIMD data layout concepts roughly correspond with column-wise versus row-wise database storage. For some operations, the AOS representation is more convenient. One example of this is compaction of selections, which introduces a data dependency, limiting SPE throughput severely. While paying this cost of selection, it is thus better to compact *multiple* SOA input columns at once using SIMD, thus producing one AOS output. An example of such a reorganization is shown in Figure 4, where we see four input vectors being combined into a single quadword vector, with each quadword containing a value from each of the four input columns, throwing away non-selected tuples in the process. Currently, we only considered packing data of the same type together. For each supported data type we can have a pre-generated `selpack` primitive, but also a `pack` (a version without selection) and an `unpack`. The query optimizer should decide the proper data layout (selection/dense, AOS/SOA).

Hash Aggregation. For the mapping of database operations to SIMD instructions, we build on [15], that described projections, selections, joins and index lookup. Lacking still in this list were grouped aggregates, which form an important part of our example query (TPC-H Q1). Grouped aggregates can be SIMD-ized if multiple aggregates of an equal type need to be computed. In case of TPC-H Q1 (see Figure 3), there are 4 floating point SUMs, one integer SUM and one COUNT (which can be treated as an integer SUM of a constant column filled with one-s).

In this case, we can pack the four float columns to be aggregated into a quadword vector `values` in AOS layout, using the `selpack` operation, and do the following:

```
vector int *grp;
vector float *values;
for (i=0; i < n; i++)
   int id = si_to_int(grp[i]);
   aggr[id] = spu_add(aggr[id], values[i]);
```

This updates four aggregate results in parallel. Here we assume that previously an aggregate group-ID has been computed and is available in AOS int vector `grp[i]`. For space reasons, we omit a detailed discussion of SIMD hashing here. SIMD-based Cuckoo hash on Cell is discussed in [14].

Platform	Evaluation Strategy		
	column at-a-time	vector at-a-time	**vector SIMD**
Itanium2 1.3Ghz	3400	311	n.a.
PPE + 1 SPE (3.2GHz)	493	459	95
PPE + 2 SPEs (3.2GHz)	280	229	47
PPE + 3 SPEs (3.2GHz)	202	153	32
PPE + 4 SPEs (3.2GHz)	178	115	24
PPE + 5 SPEs (3.2GHz)	142	92	19
PPE + 6 SPEs (3.2GHz)	129	77	16

Table 1: TPC-H Q1: avg elapsed msec (SF=1)

4.1 Evaluation: TPC-H Q1

Table 1 lists the initial results of our Cell experiments on the SF-1 TPC-H dataset (6M lineitems, RAM resident) on query 1. This query is a good measure of the computational power of a database system as it is a simple Select-Project-Aggregate query, that consumes a large input table, producing almost no output, and performs quite a few computations. Our main result is the "Vector SIMD" column that shows the (almost perfect) parallel scaling of our SIMD implementation of vectorized query processing. For comparison, we reproduce results from [3] obtained on a 1.3GHz Itanium2: Cell is 20 times faster than MonetDB/X100 on Itanium2 (16 vs 311 msec). An important requirement to obtain such speedup is proper use of SIMD friendly code. Just compiling the standard MonetDB/X100 primitives for the Cell ("vector at-a-time") is 5 times slower; but still beats the same code on Itanium2 by a factor 4. While the original MonetDB strategy of full materialization (which causes huge memory traffic on Q1) brings the Itanium2 memory subsystem to its knees (3.4sec), we see the 25.6GB/s Cell memory infrastructure holding up well (129msec), though scaling is sub-linear. From this data, we speculate that massive scatter/gather algorithms on Cell are likely to yield bandwidth-bound results. Such results might be acceptable, but certainly not optimal (129 vs 16 msec here).

5. RELATED WORK

The only paper that explicitly touches upon Cell in the context of data management (hashing, in this case) is [14], where the computational power of the SPEs is used for quick hash function computation. Our work builds strongly on [15], that describes the applicability of SIMD instruction for database workloads. We extend this work by proposing use of Array-Of-Structure (AOS) data layout to perform grouped aggregation in a SIMD-ized fashion as well. Database workloads on a network processor, which, similar to Cell SPEs, lack a hardware cache are analyzed in [6]. Both [17] and [8] try to improve instruction-cache reuse by reusing its contents on buffered data from within the same query, or on data from other queries respectively. Work on automatic SPE code partitioning and management is conducted by the IBM compiler team [5]. This compiler is as of yet not available, so for our Cell database system we created our own code management runtime.

6. CONCLUSION & FUTURE WORK

In this paper we have taken a sneak preview into a possible future of query processing on heterogeneous multi-core CPUs, by using the Cell for database purposes. We made a case for column-wise query processing on Cell, as it reduces branchiness of code, allows for better instruction locality, and produces code that is amendable to efficient (and sometimes even automatic) SIMD translation. However, we have also shown that care needs to be taken not to materialize intermediate results in main-memory, to avoid bus contention. These ideas correspond to the *vectorized query processing* model used in MonetDB/X100, of which parts were ported to enable these experiments. However, the default MonetDB/X100 primitive functions turned out to yield suboptimal SIMD translations on the SPEs. We added support for AOS (Array of Structures) vector data layout, which allowed to better SIMD-ize selection and aggregation primitives.

We experimented with a limited set of operators here, but we believe that with careful engineering of parallel algorithms, more complex operators like joins and aggregations that exceed LS capacity can benefit from the exceptional computational power of Cell as well. So far, we performed main memory resident queries only. Given its enormous throughput, it is an interesting question whether Cell can be kept in balance with secondary storage when processing data beyond main-memory.

7. REFERENCES

[1] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, 2001.

[2] P. Boncz and M. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal*, 8(2):101–119, 1999.

[3] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, 2005.

[4] J. Cieslewicz, J. W. Berry, B. Hendrickson, and K. A. Ross. Realizing parallelism in database operations: insights from a massively multithreaded architecture. In *DaMoN*, 2006.

[5] A. E. Eichenberger et al. Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine Architecture. *IBM Systems Journal*, 45(1):59–84.

[6] B. T. Gold, A. Ailamaki, L. Huston, and B. Falsafi. Accelerating database operations using a network processor. In *DaMoN*, 2005.

[7] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[8] S. Harizopoulos and A. Ailamaki. STEPS Towards Cache-Resident Transaction Processing. In *Proc. VLDB*, 2004.

[9] IBM Corporation. *Cell Broadband Engine Programming Handbook*, 2006.

[10] S. Manegold, P. Boncz, N. Nes, and M. Kersten. Cache-Conscious Radix-Decluster Projections. In *Proc. VLDB*, Toronto, Canada, 2004.

[11] S. Padmanabhan, T. Malkemus, R. C. Agarwal, and A. Jhingran. Block oriented processing of relational database operations in modern computer architectures. In *Proc. ICDE*, 2001.

[12] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *Proc. ICDE*, 2006.

[13] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[14] K. A. Ross. Efficient hash probes on modern processors. In *Proc. ICDE*, 2006.

[15] J. Zhou and K. A. Ross. Implementing database operations using simd instructions. In *Proc. SIGMOD*, 2002.

[16] J. Zhou and K. A. Ross. Buffering accesses to memory-resident index structures. In *Proc. VLDB*, 2003.

[17] J. Zhou and K. A. Ross. Buffering database operations for enhanced instruction cache performance. In *Proc. SIGMOD*, 2004.