A Many-core Architecture for In-Memory Data Processing

Sandeep R Agrawal
sandeep.r.agrawal@oracle.com
Oracle Labs

Sam Idicula
sam.idicula@oracle.com
Oracle Labs

Arun Raghavan
arun.r.raghavan@oracle.com
Oracle Labs

Evangelos Vlachos
evangelos.vlachos@oracle.com
Oracle Labs

Venkatraman Govindaraju
venkat.govindaraju@oracle.com
Oracle Labs

Venkatanathan Varadarajan
venkatanathan.varadarajan@oracle.com
Oracle Labs

Cagri Balkesen
cagri.balkesen@oracle.com
Oracle Labs

Georgios Giannikis
georgios.giannikis@oracle.com
Oracle Labs

Charlie Roth
charlie_roth@yahoo.com
Oracle Labs

Nipun Agarwal
nipun.agarwal@oracle.com
Oracle Labs

Eric Sedlar
eric.sedlar@oracle.com
Oracle Labs

ABSTRACT

For many years, the highest energy cost in processing has been data movement rather than computation, and energy is the limiting factor in processor design [21]. As the data needed for a single application grows to exabytes [56], there is clearly an opportunity to design a bandwidth-optimized architecture for big data computation by specializing hardware for data movement. We present the Data Processing Unit or DPU, a shared memory many-core that is specifically designed for high bandwidth analytics workloads. The DPU contains a unique Data Movement System (DMS), which provides hardware acceleration for data movement and partitioning operations at the memory controller that is sufficient to keep up with DDR bandwidth. The DPU also provides acceleration for core to core communication via a unique hardware RPC mechanism called the Atomic Transaction Engine. Comparison of a DPU chip fabricated in 40nm with a Xeon processor on a variety of data processing applications shows a 3× - 15× performance per watt advantage.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; **Special purpose systems**;

KEYWORDS

Accelerator; Big data; Microarchitecture; Databases; DPU; Low power; Analytics Processor; In-Memory Data Processing; Data Movement System

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123985

ACM Reference format:
Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and Eric Sedlar. 2017. A Many-core Architecture for In-Memory Data Processing. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123985

1 INTRODUCTION

A large number of data analytics applications in areas varying from business intelligence, health sciences and real time log and telemetry analysis already benefit from working sets that span several hundreds of gigabytes, and sometimes several terabytes of data[9]. To cater to these applications, in recent years, data stores such as key-value stores, columnar databases and NoSQL databases, have moved from traditional disk-based storage to main-memory (DRAM) resident solutions [1, 9, 40]. However, today's commodity hardware solutions, which serve such applications employ powerful servers with relatively sparse memory bandwidth—a typical Xeon-based, 2U sized chassis hosts 8 memory channels (4 channels per socket).

For applications which scan, join and summarize large volumes of data, this hardware limit on memory bandwidth translates into a fundamental performance bottleneck [8, 35, 49, 62]. Furthermore, several features which contribute to server power, such as paging, large last-level caches, sophisticated branch predictors, and double-precision floating point units, have previously been found to be unutilized by applications which rather rely on complex analytics queries and fixed-point arithmetic [2, 9]. The work presented in this paper captures a subset of a larger project which explores the question: How can we perform analytics on terabytes of data in sub-second latencies within a rack's provisioned power budget? In particular, this paper focuses on optimizing the memory bandwidth per watt on a single, programmable data processing unit (DPU). This scalable unit then allows packing up to ten times as many memory channels in a rack-able chassis as compared to a commodity server organization.

To identify on-chip features essential for efficient execution, we analyzed the performance of complex analytics on large volumes of data like traditional TPC-H query benchmarks, and also more contemporary text and image processing operations applied when ingesting and querying data from memory. Firstly, a large portion of the power in present systems is spent in bringing data closer to the cores via large cache hierarchies [34]. Secondly, these queries need to be broken down into simple streaming primitives, which can then be efficiently parallelized and executed [48]. Thirdly, the variety of operations performed by these queries raises the need for the cores to be easily programmable for efficiency.

We examined alternate commodity low-power chip architectures for such applications. GPUs have been increasingly popular in data centers for their significant compute capabilities and memory bandwidth [3, 4, 29, 30]. However, their SIMT programming model is intolerant to control flow divergence which occurs in applications such as parsing, and their dependence on high bandwidth GDDR memory to sustain the large number of on-die cores severely constrains their memory capacity.

We also considered FPGA based hardware subroutines to offload functions, reviewed contemporary work on fixed-function ASIC units for filtering, projecting, partitioning data and walking hash-tables ([36, 41, 61, 62]), and experimented with large, networked clusters of wimpy general-purpose cores similar to prior research[5, 38]. We found that while ASIC units could significantly reduce power over off-the-shelf low-power cores by optimizing data movement, very specific fixed function units separated from the instruction processing pipeline impeded application development.

We therefore opted to engineer a customized data processing unit which integrates several programmable, low-power dpCores with a specialized, yet programmable data movement system (DMS) in hardware. Each dpCore contains several kilobytes of scratchpad SRAM memory (called DMEM) in lieu of traditional hardware managed caches. Software explicitly schedules data transfers via the DMS to each dpCore's DMEM through the use of specialized instructions called *descriptors*. Descriptors enable individual dpCores to schedule extremely efficient operations such as filter and projection, hash/range partitioning and scatter-gather operations at close to wire-speed. Each DPU also consists of a dual-core ARM A9 Osprey macro and an ARM M0 core to host network drivers and system management services.

Each dpCore is capable of fixed-function arithmetic, full 64-bit addressability, and can communicate and synchronize with other dpCores using a custom Atomic Transaction Engine (ATE) as an alternative to hardware-based cache coherence. The extremely low power design (50mW per dpCore and 6W for the entire DPU in the 40nm node) allows us to scale this unit, attached to every DRAM channel across a whole rack. Our initial prototype consists of 1440 DPUs, each with 32 dpCores and an 8GB of DDR3 memory, providing an aggregate memory bandwidth of >10TB/s and a memory capacity of >10TB in a full-sized (42U) rack.

As noted by previous studies of analytics workloads [22], (i) execution-time is dominated by stalls in application code and operating system, with long-latency memory accesses contributing to a bulk of the stalls, and (ii) there is little sharing of data between processors. Hence we design our software runtime to schedule application code without pre-emption on the dpCores and overlap data movement via the DMS. We also abstract inter-dpCore communication and synchronization routines over the ATE to allow porting of common parallel programming paradigms such as threads, task queues, and independent loops. These runtime hardware abstractions enable large scale, efficient, in-memory execution of heavyweight applications including SQL processing offloaded from a commercial database on our prototype system.

The following sections describe our experiences with designing, fabricating, and programming the DPU chip. In particular, we highlight the following contributions:

- The low-power hardware architecture of the DPU (Section 2)
- The microarchitecture and software interface of our novel data movement system (Section 3)
- The ISA extensions and programming environment which allow relatively straightforward software development on the DPU (Section 4)
- Example parallel applications which achieve 3× - 15× improvement in performance per watt over a commodity Xeon socket when optimized for the DPU hardware (Section 5).

2 DPU ARCHITECTURE

The primary goal of the Data Processing Unit (DPU) is to optimize for analytic workloads [22, 47, 58] characterized by (i) large data set sizes [63] (in the order of tens of TB), (ii) data parallel computation, and (iii) complex data access patterns that can be made memory bandwidth-bound using software techniques. A typical analytic workflow involves partitioning the working set into several data chunks that are independently analyzed, followed by a final result aggregation stage [47]. Hence an ideal architecture for such a memory-bound workload would aim to compute at memory-bandwidth.

Such computation at memory bandwidth requires keeping the workload memory-resident, with DRAM memory channels feeding data into the processing units. With a practical memory channel bandwidth of 10 GBps (DDR3-1600), to scan a nominal workload size of 10 TB in under a second, we require ≈1000 channels per rack. This results in 3KW (at 3W per channel) budgeted for main memory leaving only 17W (considering 20KW per rack [5]) for the rest of the system (per channel) including networking between the DPUs. A standard PCIe controller consumes a minimum of 10W, leaving a power budget of < 7W for the processor.

Two principles therefore guide the DPU's architecture: (i) specialize hardware to optimize for data movement, and (ii) replace power-hungry hardware features which are not performance critical for data parallel applications with low power hardware assists which allow software to implement the feature.

2.1 Data Movement

Efficiently moving data from memory to the computation unit is a key challenge for our data-intensive workloads. Commodity processors utilize hardware prefetchers to keep data close to cores. Besides being large and power-hungry, such structures may also fail to learn irregular data access patterns [22]. Instead we utilize a software-programmable data movement engine called Data Movement System (DMS) in conjunction with a small (32KB) software

managed scratchpad SRAM called *DMEM* to feed the processing cores. Unlike conventional DMA engines, the DMS supports complex access patterns that involve data partitioning and projection while transferring data. Although the idea of using specialized data movement engines particularly for partitioning data has been proposed in the past [61], our proposed DMS directly places data in DMEM making it immediately available for consumption for the processing cores.

A task running on the data processing unit programs the DMS using a simple data movement specification called a *descriptor* which instructs the DMS engine to move data in and out of the DMEM. The power of DMS descriptors is exemplified by the fact that 16MB of data can be streamed through a DMEM of 32KB at line speeds with just three DMS descriptors (detailed in Section 3). Our experiments achieve a maximum bandwidth greater than 10 GBps—near peak bandwidth on a DDR3 channel.

2.2 Data Processing

With energy-intensive data movement offloaded to the DMS, the data processing cores (called dpCore) demand a simpler, low-power design. The dpCore features a 64-bit MIPS-like ISA for general purpose compute. To accelerate common analytic query operations like filters and joins, the ISA provides single-cycle instructions like bit-vector load (`BVLD`), filter (`FILT`) and CRC32 hashcode generation. For example, in conjunction with efficiently loading data into DMEM, the `BVLD` and `FILT` instructions could be repeatedly used to efficiently filter through a sparsely populated column. These instructions help with accelerating common data summarizations like population counts and scatter-gather masks.

The dpCore implements a simple dual-issue pipeline, one for the ALU and the other for the LSU pipe. The ALU supports a low-power multiplier that stalls the pipeline for multiple cycles and has no native support for floating point arithmetic. In addition, the dpCore uses a simple conditional branch predictor that predicts backward branches as taken. The memory model is relaxed, with instructions to fence-off pending loads and stores.

The dpCore has no memory management unit and programs directly address physical memory. Hence all programs running on the dpCore share the same address space. To support basic software debugging and simple address space protection, the dpCore provides a few instruction and data watchpoint registers that raise an exception on any address boundary violation.

Overall, there are 32 low-power dpCores organized into 4 macros (shown in Figure 1) where some DMS logic private to the dpCores (shown as DMAD in the figure) is replicated along with the dpCores. The 32 dpCores cooperatively work on large datasets in DRAM exploiting any data-parallelism. The 4 dpCore macros together with the DMS form the dpCore Complex that does the bulk of all the work in the larger SoC (more details in Section 2.4).

2.3 On-Chip Communication

Although the majority of a workload's data accesses go through the DMEM using the DMS, the dpCore also supports a general-purpose cache hierarchy, which includes core-private 16KB L1-D and 8KB L1-I caches and a 256KB last level (L2) cache shared between dpCores in a macro. To reduce chip complexity and power, hardware does not manage coherency between caches. Instead, the ISA provides cache flush and invalidate instructions to enable software-managed coherence.

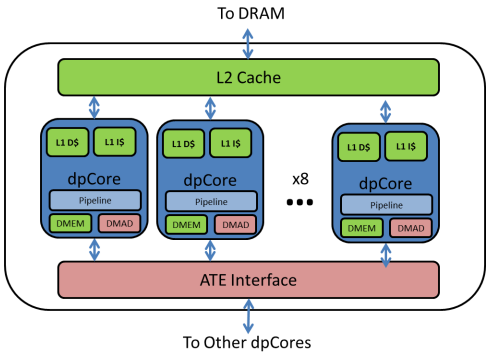

Figure 1: Block diagram of a dpCore Macro

A hardware block called the Atomic Transaction Engine or ATE allows communication between the dpCores. The ATE block comprises of a 2-level crossbar—one crossbar connecting 8 dpCores in a macro and one between the 4 macros, and hardware to manage messaging with guaranteed point-to-point ordering over this interconnect. The ATE software registers space in each dpCore's DMEM for use by the ATE hardware block. On each dpCore, the hardware ATE engine manages the DMEM pointers and delivers messages and interrupts.

The ATE hardware interprets certain messages as remote procedure calls to be performed by hardware on the receiving dpCore. The message types and payload can request for a load, store, atomic fetch and add, or an atomic compare-and-swap operation to be performed on any address in DDR or DMEM space at the remote dpCore (ATE Hardware RPCs). Upon receipt, the ATE engine in the remote core decodes and injects the operation in the dpCore pipeline. Although such an operation appears as stalls in the remote dpCore's instruction stream, it does not generate an interrupt or perturb the instruction cache. The ATE supports more complex atomic operations in the form of software remote procedure calls (ATE Software RPCs). When the ATE hardware dequeues messages of this type, it interrupts the software on the remote core and jumps to a pre-installed software handler which then executes to completion. Hardware RPCs are similar to x86 atomics, except that they allow a dpCore to operate on another dpCore's DMEM directly.

Hardware RPCs enable efficient synchronization primitives such as mutexes and barriers, while software RPCs allow an environment to flush, invalidate and mutate shared address ranges. For remote procedure calls which expect return values (such as fetch-and-add), the ATE hardware ensures atomicity, FIFO ordering through the interconnect, and stalls the requesting dpCore until the value is received. Software on the dpCore may issue one outstanding ATE request at a time, after which it can choose to process regular instructions before eventually blocking for response from the ATE hardware. Figure 2 shows measured response times for typical ATE requests; by scheduling independent instructions for such duration between ATE requests and waiting for response, software can optimize for throughput.

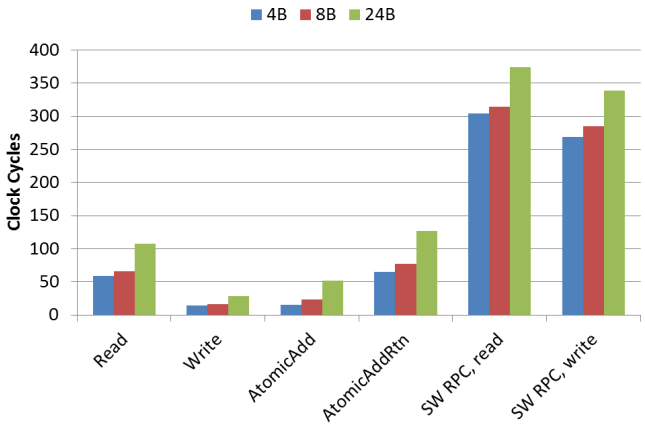

Figure 2: Performance of ATE remote procedure calls

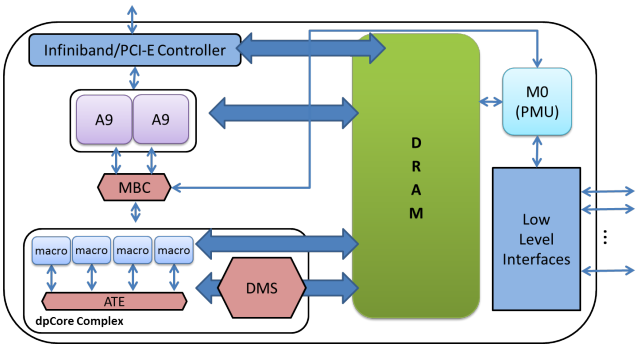

Figure 3: Block diagram of a DPU

2.4 SoC Organization

Putting it altogether, the DPU is a System-on-a-Chip that features the above described low-power specialized dpCore complex optimized for data-parallel analytic workloads. In addition to the dpCore complex, the SoC also includes a Power Management Unit (PMU or M0), an ARM Cortex-A9 dual core processor, a MailBox Controller (MBC) and some peripherals (including PCIe and DRAM controllers). A schematic depiction of the DPU SoC is shown in Figure 3.

The M0 processor manages the dpCore's power modes (supports 4 states) and enables power gating of individual dpCore macros. The A9 processors serve as a networking endpoint and provides a high bandwidth interface to peer DPUs by running an Infiniband network stack on Linux.

The MBC is a hardware queue [37, 54, 59], providing a simple communication interface that connects the dpCores, A9 cores and the M0 processor. Its goal is to facilitate quick exchange of light-weight messages (i.e., sending a pointer to a buffer in memory), while the bulk of the data is communicated through main memory. It maintains a total of 34 mailboxes, one for every dpCore, one for the A9 cores and one for the M0. The MBC maintains a set of memory mapped (RD/WR) control and data registers for each mailbox that can be used to send (WR) and receive (RD) on that mailbox. Each mailbox also controls an interrupt line that is used to notify the corresponding destination core on arrival of a message.

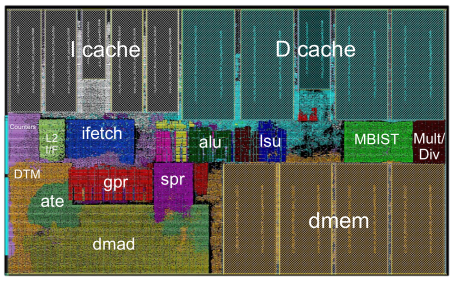

Figure 4: dpCore processor implementation

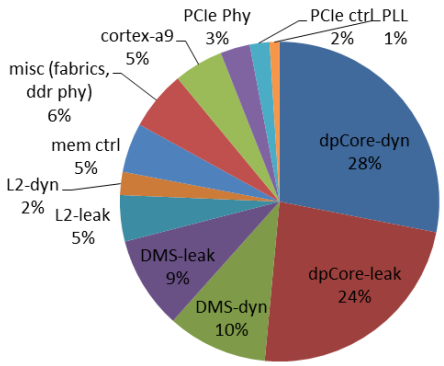

Figure 5: DPU power breakdown (Total power = 5.8 W)

2.5 Fabrication

We fabricated the DPU using a 40 nm process, with a silicon area of 90.63 mm^2 and 540 millions transistors, of which 268 million transistors are used for memory cells. Figure 4 shows the implementation of a single dpCore. We went through an extensive physical design and verification process, the details of which are beyond the scope of this paper. We used formal verification techniques as well, and almost 16% of our RTL bugs were found via formal tools. We design the DPU for a provisioned power of 5.8W, and Figure 5 shows the power breakdown of the 40 nm DPU from the post silicon flow. Over 37% of our power goes towards leakage, since we use high leakage circuits to meet timing constraints. Each dpCore consumes 51 mW of dynamic power at 800 MHz, highlighting our focus on low power design and efficiency. We optimize this design for provisioned power, not dynamic power, since our aim is to minimize rack-scale provisioning costs.

We also designed a variation of the DPU architecture for the 16 nm process node. This process shrink allows us to increase dpCore density to 160 dpCores on a die, and the number of transistors to 3 Billion. To avoid additional design costs, we replicate 5 copies of the existing 32 dpCore complex on the new DPU. These complexes share an upgraded DDR4-3200 main memory unit providing 76 GB/s of memory bandwidth/DPU allowing us to maintain our memory-compute design point. This also increases the TDP of a

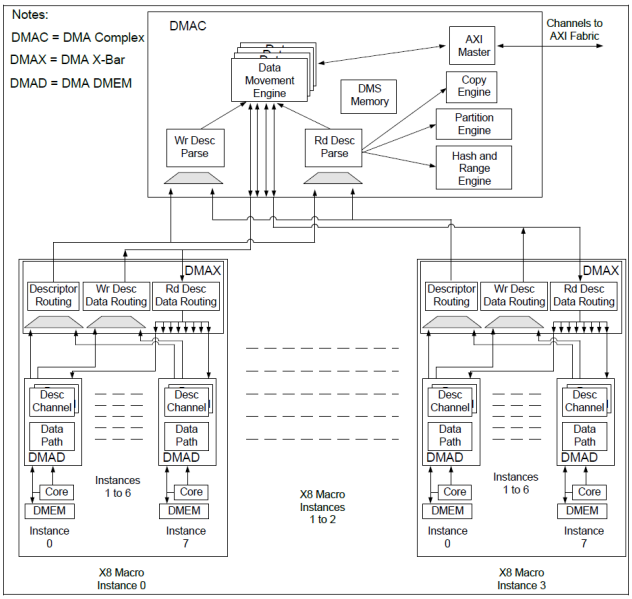

Figure 6: Data Movement System (DMS)

DPU to 12W, however with a 5× increase in compute and memory bandwidth, each DPU becomes 2.5× more efficient in terms of performance/watt.

3 DATA MOVEMENT SYSTEM

DMS is the cornerstone of the DPU that is critical to achieving our design goals: 1) optimize for data movement at memory bandwidth and 2) low power design. Particularly, in this section we will highlight three main characteristics of DMS that directly stem from these design principles. First, the DMS is designed to accelerate common data movement in analytics processing like scanning, data partitioning and projection (Section 3.1). Second, the DMS microarchitecture is designed to fully utilize available system memory bandwidth using well-known techniques like FIFO flow control, buffering and pipelining (Section 3.2). Finally, the DMS exposes this advanced data movement functionality with a novel software interface (Section 3.3) that completely decouples the core from data movement with infrequent and low overhead interactions. Similar to prior work [61, 62], the DMS is hence able to accelerate common data analytics operations (Section 3.4).

3.1 Architecture Overview

The DMS is a specialized hardware unit that directs data transfer between DDR memory and DMEM attached to each dpCore. Internally, the DMS hardware implements several functions to interpret memory traffic as fixed-width tuples: (i) stride over, scatter, and gather data across DMEM into and from contiguous DRAM address ranges, (ii) partition data into different dpCore's DMEMs based on programmable hash, radix and range comparisons , (iii) buffer and move intermediate results in internal SRAM memories, (iv) perform flow control and manage portions of DMEM without intervention from the dpCores. Software programs these functions by issuing

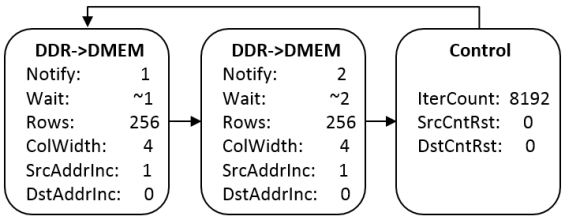

Figure 7: DMS descriptor chain used in the DMS example.

commands to the DMS in the form of 16B DMS descriptors, that enable pipelining movement and restructuring of data while running independently of more complex operations executing on the dpCores. The DMS architecture organization is shown in Figure 6.

```
dms_descriptor* desc0 = dms_setup_ddr_to_dmem(256,
     src_addr, dest_addr, event0);
dms_descriptor* desc1 = dms_setup_ddr_to_dmem(256,
     src_addr, dest_addr + 1024, event1);
dms_descriptor* loop = dms_setup_loop(desc0, 8191);
dms_push(desc0);
dms_push(desc1);
dms_push(loop);
count = 0; buffer_index = 0;
dms_event events[] = {event0, event1};
do {
  dms_wfe(events[buffer_index]);
  consume_rows();
  clear_event(events[buffer_index]);
  buffer_index = 1 - buffer_index; // toggle index
} while (++count != 16384);
```

Listing 1: DMS Programming Example

DMS Interface and Execution Model. Software constructs the descriptor in DMEM and issues a push instruction identifying the DMEM pointer and one of two DMS channels on the dpCore's DMS interface (typically segregating read and write operations). A hardware unit called the DMAD per-dpCore (DMA DMEM unit) enqueues descriptors on to an active list per channel. The DMAD links (i.e., chains) together descriptors issued on the same channel. Software may issue a special *loop control* descriptor to point back to a previous descriptor and indicate a fixed iteration count. The DMAD manages such descriptor lists and loops without intervention from dpCores. It also has source and destination address registers to support auto-increment functionality within DMS loops (refer example). Descriptors from each of the 32 read and write active lists then arbitrate via a crossbar (DMAX) into a central DMA controller (DMAC). Read and write engines in the DMAC schedule DDR transfers over a standard 128-bit AXI interface. Data received from DDR may be transfered into a receiving DMEM via the DMAX (in case of read or gather). The DMAC also performs address calculations (e.g., source and destination increments and wrap-arounds) to enable successive data transfers. After completing the data transfer the DMS signals back to the dpCore using a novel *asynchronous* event notification interface. The software checks for any such outstanding event by using a Wait-For-Event or `wfe` instruction.

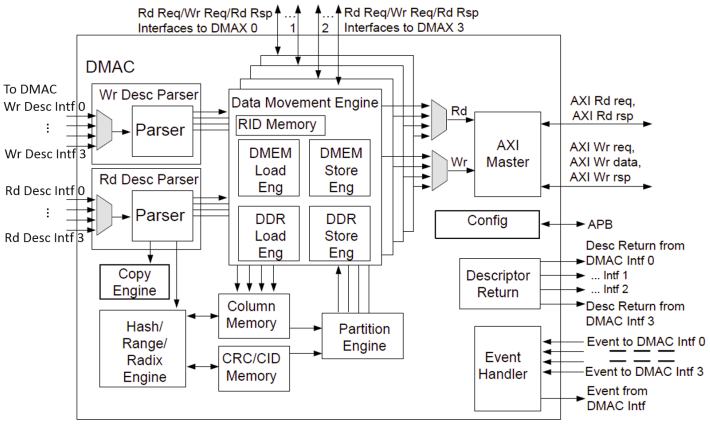

Figure 8: DMAC Block Diagram.

Data Movement Direction: Source→Dest.	DMS Operations						Main Purpose
	Scatter	Gather	Stride	Partition	Key	LastCol.	
DDR↔DMEM	X	X	X				Direct data read/write to/from memory
DMS→DMS							Move data between the DMS internal memories
DMS→DMEM				X		X	Partition pipeline and store to DMEM
DMEM→DMS		X					Transfer RID/BV data for Scatter/Gather
DDR→DMS		X			X		load key/data for partitioning
DMS→DDR	X						Store hash/CID memory to DDR

Table 1: DMS Data Descriptor Types and Supported Operations.

To help demonstrate the utility of the DMS interface, we will use a simple data move from DDR to DMEM as a running example. The program shown in Listing 1 transfers 16MB of contiguous data from DRAM to DMEM. The two DMS descriptors (`desc0` and `desc1`) are programmed to write 256 4B elements into DMEM on execution, with each descriptor identifying a unique buffer in DMEM. The DMAD links the descriptors and the loop descriptor completes the chain (Figure 7), enabling consecutive iterations to operate on alternate buffers with auto-increment source address, while the dpCore is free to consume the buffer filled in the previous iteration.

DMS Partitioning. To do complex data movement like range partitioning, data from DDR can alternatively be buffered internally in the DMAC in specially dedicated SRAM banks called *column memories* (CMEM). Descriptors issued to the DMAC can further process data in column memories. A hash and range engine can apply a CRC32 checksum to the elements of the column memories and stage the result in another dedicated internal memory called *CRC memory*. The engine can be programmed to inspect radix bits of the resulting hashed column (or alternatively the original key column) and generate a dpCore ID for each result (hash radix partitioning). The DMAC can also generate dpCore IDs by matching each column memory item against one of 32 pre-programmed ranges (range partitioning). The dpCore IDs thus generated are also stored in dedicated SRAM banks (CID memory). As a final stage in partitioning, descriptors instruct the DMAC partitioning and store engines to write the resulting data into specified locations in each dpCore's DMEM. A fourth class of internal memory is dedicated for storing bit vectors which are typically used as scatter-gather masks when moving data. In all, the DMAC has about 42.5 KB of dedicated SRAM, banked so that the internal pipeline may be fully utilized (more later). Figure 8 shows the organization of the DMAC unit.

Flow control and synchronization. The DMS associates with each dpCore a list of 32 binary events. Descriptors typically encode the setting or clearing of a particular event to signal waiting (pre-condition) and notification (post-processing). In the example, `desc0` and `desc1` are associated with `event0` and `event1` respectively to signal to the dpCore that the corresponding DMEM buffers have been filled. The DMAX supports FIFO flow control between the DMS to DMEM buffers which to coordinate the data transfer. On rate mismatch—for example if the dpCore is unable to process buffers at DRAM bandwidth, the DMAC hardware thus applies back pressure to restore flow control.

From the DMAC interface, a maximum of 4 descriptors may be outstanding to the DMAC at any instant. On the DRAM interface, the AXI bus provides 128-bit read and write data paths, and a maximum of 256B can be requested per transfer request. Hence larger DMS transfers are broken by the DMAC into multiple AXI transactions.

3.2 DMAC Microarchitecture

The three core operations of the DMAC—(a) loading from DDR to DMS memory, (b) hashing and computing core/partition IDs, and (c) storing partitions from DMS memory to DMEMs, are pipelined in hardware to allow for maximal throughput. The DMAC receives descriptors from DMAD via one of the four DMAX complexes (one DMAX per macro). Hence there are four load/store engines in the DMAC. To support scatter and gather in parallel, the internal bit vector memory is hence also banked four ways (4KB per bank). To fully sustain the three stage pipeline, the load/store engines stage the column data in three banks of column memory (each bank is 8KB).

The hash engine computes the hash on the column memory and writes to the CRC memory, while the radix stage computes core IDs based on the contents of the CRC memory. Double-buffering the CRC memory in two separate banks (each bank is 1KB) allows these stages to proceed in parallel. Finally, the CID memory is also double buffered (256B per buffer) to allow the DMS to create and consume partitions in parallel. Figure 9 illustrates how the hash partition pipeline is implemented. Although this organization

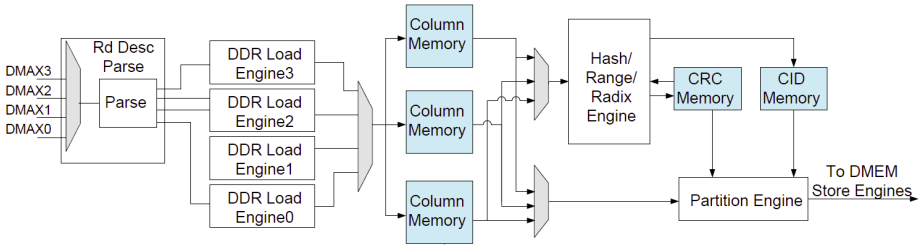

Figure 9: DMS Hash Partitioning Pipeline in DMAC.

Word0	Word1	Word2	Word3
Type[31:28], Notify[25:21], Wait[20:16], LinkAddr[15:0]	ColWidth[30:28], GatherSrc[25], ScatterDst[24], RLE[23], SrcAddrInc[17], DstAddrInc[16], DDRAddr[3:0]	Rows[31:16], DMEMAddr[15:0]	DDR Addr[35:4]

Table 2: Layout of DDR to DMEM Data Descriptor.

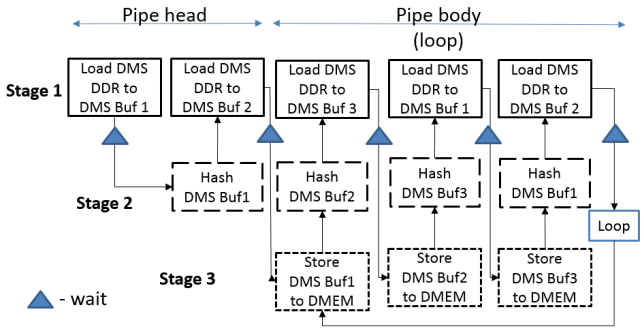

Figure 10: Three Stage Pipelining of DMS Hash Partition Operation: Here each key and data columns are loaded into one of the three column memories (one per stage), with appropriate event wait control descriptors to do flow control.

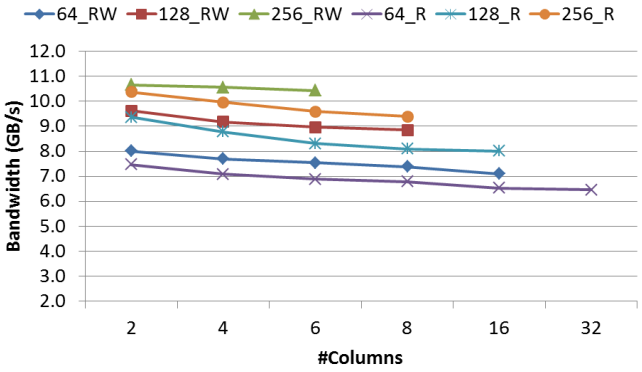

Figure 11: Bandwidth achieved across 32 dpCores for reading and reading+writing data via the DMS.

makes pipelining partitioning possible, the rich interface of the DMS descriptors is essential for exposing it to the software.

3.3 DMS Descriptors

The DMS interface is designed to enable a complex software pipeline that fully overlaps stages of compute over data movement stages in analytics applications. As mentioned earlier, DMS descriptors are macro instructions that compactly encode the DMS operations, source address, destination address and other control information such as events. There are two classes of DMS descriptors: data and control. The data descriptors, as the name suggests, help encode source/ destination address and data operations among others. Table 1 lists all types of data descriptors and supported operations in the DMS. An example DDR→DMS data descriptor layout is shown in Table 2. The control descriptors help program loops (as shown in the example), hash and range engine, and also to provide complex event operations like waiting (or set) on one or more events.

To illustrate the richness of the descriptor interface, a programmer can pipeline the hash partitioning operation (pictorially represented in Figure 10) using a combination of data and control descriptors. For lack of space the complete pseudocode is not shown here. We show that this pipelining ability achieves our ultimate goal of supporting hash partitioning at DDR memory bandwidth in the next section.

3.4 DMS Performance

In this section we highlight the raw performance and efficiency of DMS in doing common data analytics operations by using appropriate microbenchmarks.

DMS Read and Write Bandwidth. We start with measuring the read and write performance achieved using the DMS. Each dpCore reads (R) and reads/writes (RW) a table with 4K rows in memory stored in column-major format. We measure the achieved DMS bandwidth across all dpCores by varying the number of columns per row (1-32), size of each column (1, 4, 8B) and tile size in DMEM (64, 128, 256B) used by the dpCores to R/W.

Figure 11 shows the results of this experiment (for column width = 4B). For brevity, we do not show other column widths as they

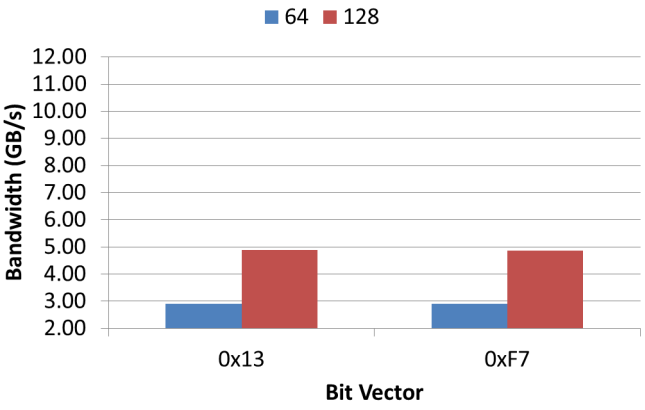

Figure 12: Bandwidth achieved across 32 dpCores for the bitvector gather operation using the DMS.

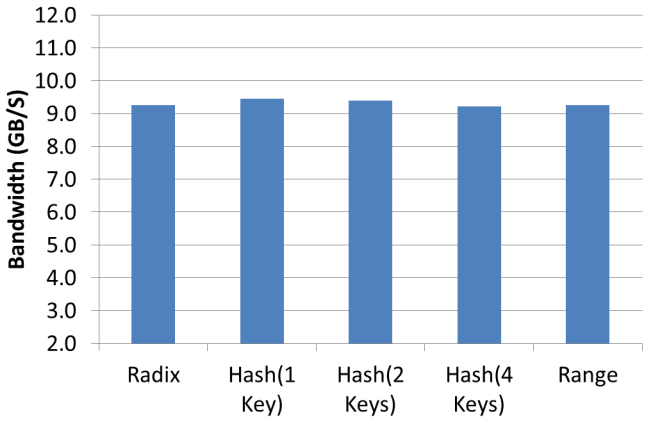

Figure 13: Bandwidth achieved with DMS partitioning engine

show similar trends. We would like to call out three interesting observations. First, we observe a slight decrease in bandwidth as the number of columns increases. As DMS fetches one column at a time, it observes a small latency overhead in fetching non-contiguous DRAM pages. Second, large buffer sizes amortize fixed DMS configuration overheads resulting in higher bandwidths. Finally, DMS achieves a bandwidth of > 9 GB/s for a buffer size of 8 KB(128 rows/buffer, 4 columns, 4B column widths) which is about 75% of the peak DDR3 bandwidth, and this is also the bandwidth we observe in many real applications (Section 5).

DMS Gather. We test a common analytic usecase that require the DMS's gather functionality. Here, we program the DMS to gather rows from DRAM to DMEM corresponding to set bits in a dense (0xF7) and a sparse (0x13) bitvector. The DMS is designed to perform gather at line speeds, however, due to an RTL bug, the first version of our chip could not utilize the DMS to its full potential (results shown in Figure 12). In brief, when all 32 cores issue gather operations, a FIFO that holds the bitvector counts in the DMAC overflows causing the DMAD units to stall indefinitely. We use a software workaround that ensures only a single dpCore issues a gather operation at a time, hence the low gather bandwidth.

Hardware Partitioning. As mentioned earlier, data partitioning is an integral part of any large-scale analytics. To measure the achievable bandwidth during DMS partitioning, we use a microbenchmark that uses the DMS to do a 32-way partition of an large input relation with four 4B columns. As before, the table is stored in the column-major format. We program the DMS to fully utilize the three stage pipeline illustrated earlier.

Figure 13 shows the effective bandwidth achieved with different partition schemes available in the DMS. Radix partitioning uses 5 bits from a key column to partition the data into 32 ways. In all the partitioning schemes, the DMS achieves 9.3 GB/s and outperforms the previous published state-of-the-art hardware accelerator for partitioning [61], where the partitioning throughput for a 32-way partitioning is 6 GB/s. In fact, as the DMS helps decouple partition completely from the dpCores, we can sustain a 9 GB/s for an additional 32 way software partition in parallel (i.e. a 1024 way partitioning). Note at higher power budget conventional systems can achieve throughput higher than 9 GB/s using other techniques like multi-threading or SMT [49].

4 SOFTWARE SYSTEM

The 32 dpCores, ARM cores, and firmware, each implement ISAs targeted by off-the-shelf compilers (gcc cross-compiled on a familiar development platform in case of this article). Each dpCore executes the same binary executable image, linked with common system utilities for hardware abstraction and concurrency primitives (for example, remote cache flush/invalidation, atomics). Applications are co-operatively scheduled to completion: only occasional interrupts from a well-known set of sources (software remote procedure calls via ATE, network messages over the mailbox, or a timer) cause control to temporarily switch away from the application thread. A two-level heap allocator similar to Hoard or TCMalloc [11, 24] allows efficient, dynamic management of most of DRAM space.

```
void* dpu_serialized(core_id_t _id, void(*rpc)(void*),
     void* args, visitor_fp args_visitor, visitor_fp
     return_visitor);
```

When programming the non-coherent system, developers need to be conscious about data placement, sharing patterns and data ownership at any given point of execution. As a programming practice, most shared data structures are pinned to a single owner dpCore, and all manipulators are forced via a serialized interface to the ATE's remote procedure calls (see listing). The programmer identifies the memory region pointed by any argument or return parameters (visitors); the underlying software: (a) flushes the argument objects on the issuing core, (b) invalidates the same on the remote core, (c) invokes the RPC with the function (the shared data manipulator) on the remote dpCore, (d) flushes the return address objects on the remote core, and (e) invalidates the remote regions when the RPC returns to the sending dpCore. The use of common/-physical address pointers on all cores (data as well as functions) allows concisely encoding all information in the ATE message.

We developed debugging tools that identify data races and coherence violations, ranging from simulator extensions that monitor

Workload	Domain	Applications
Support Vector Machines	Machine Learning	Classification problems in healthcare [64][52], document classification [32], handwriting recognition [7]
Similarity Search	Text Analytics	Web search[13], Image Retrieval[12, 55]
SQL Operations	SQL Analytics	Queries on Structured Data
HyperLogLog	NoSQL Analytics	Spam Detection[10], Mining of massive datasets[25], Database operations (COUNT DISTINCT)
JSON Parsing	NoSQL Analytics	Dynamic webpages[60], Web services, Data exchange formats
Disparity	Machine Vision	Robotics[44], Pedestrian tracking[26]

Table 3: Our list of DPU Applications

code execution at instruction level to a static binary instrumentation tool that monitors code execution on the DPU at runtime. We modified the compiler to align each global variable to cache-block boundaries to avoid false sharing. Programmers tended to conservatively flush/invalidate to avoid coherence errors which penalized performance; we hence developed a tool to identify and quantify redundant cache operations.

Apart from the dpCores, the firmware and ARM cores run essential services such as network drivers (PCIe and Infiniband to message other DPUs, mailbox driver to message dpCores or other ARM cores within the same DPU), and system health reporting (e.g., thermal trips, memory faults). Within each DPU SoC, less than 1GB of DRAM suffices to host the operating systems, driver, and user code for the ARM cores, and the binary executable for the dpCores. Such system services allowed us to scale several of the applications in Section 5 across 500+ DPU clusters.

5 CO-DESIGN APPLICATIONS FOR DPU

We designed the DPU to be able to perform in-memory analytics at peak memory bandwidth and corresponding power efficiency, and we look at applications spanning a variety of domains (Table 3) on our hardware. In this section, we describe how we used the unique features of our hardware to implement each of these applications efficiently; for comparison, we use x86 implementations that employ state of the art algorithms.

We compare our numbers to a Xeon server, with two Intel Xeon E5-2699 v3 18C/36T processors and 256GB DDR4 DRAM running at 1600 MHz. For our DPU experiments, all datasets were converted to 10.22 software fixed point. There has been a increasing amount of research on using fixed point for machine learning algorithms [28][17], and we observed negligible loss in accuracy while comparing with a floating point implementation. A simple 10.22 fixed point approach works due to the fact that most machine learning algorithms require data normalization, which constrains the range of the numbers involved, leaving 22 bits to handle precision. To compute performance/watt, we assume a TDP of 145W for the Xeon, and 6W for the DPU. Figure 14 shows the performance/watt gains of the 40nm implementation of the DPU for applications that we looked at normalized to an optimized x86 implementation. The power numbers correspond to provisioned SoC power for both the DPU and x86.

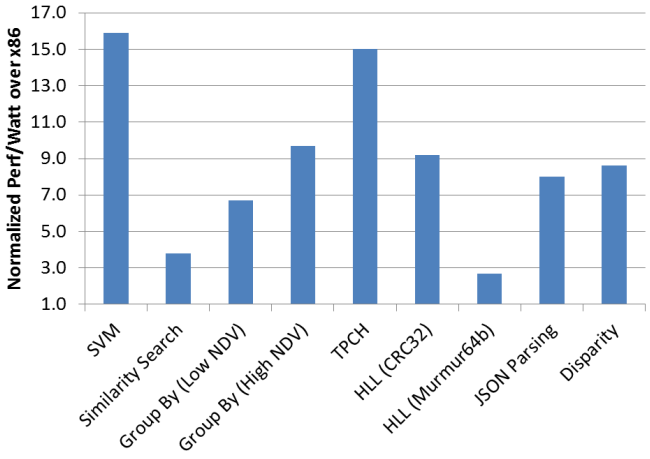

Figure 14: DPU Efficiency gains for several applications

5.1 Support Vector Machines

Given a set of training samples, the Support Vector Machine (SVM) problem aims to learn an *optimal* decision boundary in the sample space by iteratively solving a convex quadratic programming problem. We implement a variation of the Parallel SMO algorithm proposed by Cao et. al [14] on the DPU. Each iteration of the SMO algorithm involves computing the *maximum violating pair* across all training samples, and the algorithm converges when no such pair could be found. We distribute the computation of the maximum violating pair across all dpCores, and each core sends its *local violating pair* to a designated master core using the ATE. The master then computes the error on the global pair, and broadcasts the updated values to all dpCores using the ATE as well. We use the DMS to read and write the samples and coefficients arrays at line speeds, further improving efficiency.

We compare the DPU version with a multicore LIBSVM [16] implementation on x86. We use 128K samples from the HIGGS [39] dataset for evaluation. Optimal parameters are chosen for LIBSVM (100MB kernel cache, 18 OpenMP threads) empirically. The DPU version generates kernels on the fly, since we found generating and maintaining a kernel cache for the entire dataset to be much slower. A side-effect of our fixed point implementation is that the DPU converges in 35% fewer iterations, with no loss in classification accuracy, while being over 15× more efficient than LIBSVM.

5.2 Similarity Search on Text

The similarity search problem involves computing similarities between a group of queries and a set of documents indexed using the *tf-idf* scoring technique, and coming up with *topk* matches for each query. Computing cosine similarities for a group of queries against an inverted index of documents can be formulated as a Sparse Matrix-Matrix Multiplication problem (SpMM)[3]. We leverage recent research on optimizing SpMM on the CPU [46] and the GPU [3] and implement these algorithms on x86 and the DPU. Each query independently searches across the index, making the problem easily parallelizable across multiple threads/dpCores. We

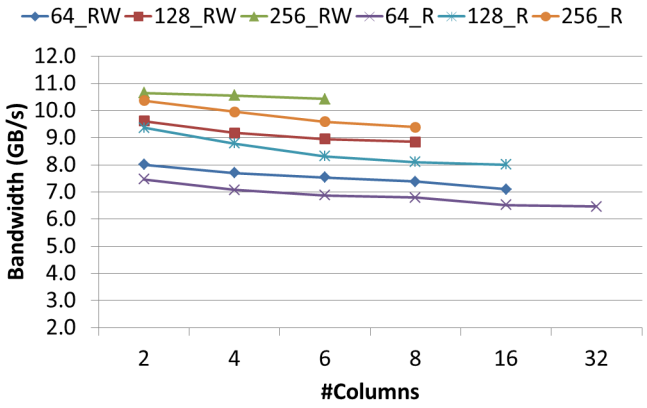

Figure 15: Performance on a dpCore for the filter primitive

search across 4M pages in the English Wikipedia, using page titles as queries, similar to [3].

A SpMM operation ($C = A \times B$) relies on a simple principle, accumulate rows of B corresponding to non-zero columns of A into C. A and B are stored in the Compressed Sparse Row (CSR) format. Contemporary SpMM algorithms [3][46] rely on tiling (range-partitioning) B and C, allowing working sets to fit in the LLC. The CSR format makes DMS access to a tile challenging, since we cannot know when a tile ends without actually reading the tile. Naively using the DMS involves fetching a buffer containing a tile, utilizing the tile, and discarding the rest of the buffer. This generates an effective bandwidth of only 0.26 GB/s across 32 dpCores. We use a novel technique for SpMM, where we fetch a buffer containing multiple tiles into DMEM, and track state corresponding to the end of each tile. Dynamically forming tiles allows us to consume all data in DMEM, improving the effective bandwidth to 5.24 GB/s on the DPU and a 3.9× improvement in performance/watt over a optimized Xeon implementation (effective bandwidth across 36 cores - 34.5 GB/s).

5.3 SQL Operations

We implement a SQL processing engine on the DPU, and use it to benchmark several common analytic operations as well as some TPCH queries. We omit the design of this engine for brevity, and focus on architectural features of the DPU that allow us to accelerate this class of applications.

Filter. This is a basic SQL operation used to select rows that satisfy a given condition. In our evaluation of filter, we program the DMS to fetch a single column of data, and vary the tile size. The DMS fetches a tile of the requested size into DMEM; double-buffering in DMEM is used to pipeline this with the dpCores' execution of `BVLD` and `FILT` instructions to generate a bitvector representing the rows that satisfy the condition. A single dpCore achieves a bandwidth of 482 Mtuples/second (Figure 15), which translates to 1.65 cycles/tuple, and a peak memory bandwidth of 9.6 GB/s for 32 dpCores.

Grouping and Aggregation (SQL Group-By). This SQL operation consists of grouping rows based on the values of certain columns (or, more generally, expressions) and then calculating aggregates (like sum and count) within each group. It can be efficiently processed using a hash table as long as the number of distinct groups is small enough [20]. Since the access pattern is random and the hash table size grows linearly with the number of distinct groups, ensuring locality of access is very important for performance, especially on the DPU architecture.

Our query processing software is designed around careful partitioning of the data to ensure that each partition's data structures (like a hash table, in the case of group-by) fit into the DMEM. This also guarantees single-cycle latency to access any part of the hash table, unlike a cache.

The process begins with the query compiler where the DMEM space is allocated among input/output buffers, metadata structures and the hash table in a way that maximizes performance. Typically, each input/output buffer doesn't benefit much from more than 0.5 KB and hence a large part of the DMEM space is allocated to the hash table. Then the number of partitions needed to achieve that hash table size per partition is calculated. The partitioning needs to be performed using a combination of hardware and/or software partitioning. Software partitioning internally uses DMEM buffers for each partition and column; so, based on the number of columns involved, we can calculate the maximum number of software partitions that can be achieved in one "round" (round-trip through DRAM, reading data in and writing it out as separate partitions) at a rate that is close to memory bandwidth. The number of rounds of partitioning required is then calculated and partition operators are added to the query plan before the grouping operator.

At runtime, if the size of a partition is larger than estimated, the execution engine can re-partition the data for that partition as needed. In the last round, if the number of partitions is less than the number of cores, only hardware partitioning is needed; this is especially useful for moderately sized hash tables (which are larger than DMEM but not larger than the combined size of all the cores' DMEM) since no extra round-trip through DRAM is needed.

Partitioning also provides a natural way to parallelize the operation among the cores, since each core can usually operate on a separate partition. But when the number of distinct groups is low, partitioning is not necessary or useful; in this case, the input data is equally distributed among the cores and a merge operator is added to the query plan after the grouping operator. Since the merge operator only works on aggregated data, its overhead is very low.

We evaluate Group-by for both low and high number of distinct values (Low-NDV and High-NDV cases in Figure 14). In the Low-NDV case, both x86 and DPU platforms are able to process the operation at a rate close to memory bandwidth; so the improvement (6.7×) is primarily due to the DPU's higher memory bandwidth per watt. However, in the high-NDV case, the data needs to be first partitioned on both platforms. Due to the DMS's hardware partitioning feature the DPU only needs to do one round of partitioning, whereas x86 needs two rounds; so the improvement (9.7×) is higher in this case.

TPCH Queries. We also implemented other SQL operations like Join and Top-k using partitioning techniques similar to those described above. We connected our SQL engine running on the DPU to a widely used commercial database with in-memory columnar

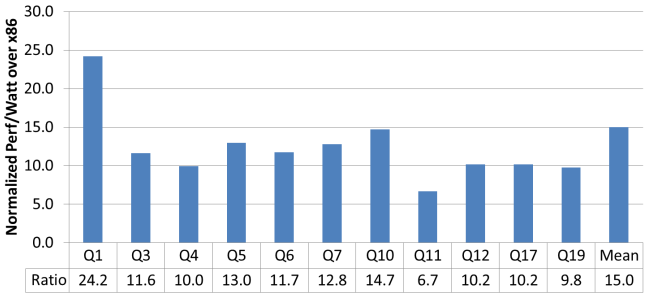

Figure 16: DPU Efficiency gains for TPCH queries

query execution capability on X86 and offloaded the execution of SQL queries from the database to the DPU. We compared the performance of TPCH queries running on the DPU to X86. We achieved an overall (geometric mean) improvement of 15× (Figure 16) in terms of performance/watt over x86.

5.4 HyperLogLog

The HyperLogLog (HLL) algorithm [23] provides an efficient way to approximately count the number of distinct elements (the cardinality) in a large data collection with just a single pass over the data. HyperLogLog relies on a well behaving hash function which is used to build a most likelihood estimator by counting the maximum number of leading zeros (NLZ) in the hashes of each data value. This estimation is coarse-grained, and the variance in this approach can be controlled by splitting the data into multiple subsets, computing the maximum NLZ for each subset, and using a harmonic mean across these subsets to get an estimate for the cardinality of the whole collection. This also makes the algorithm easily parallelizable, each core computes the maximum NLZs for its subsets, followed by a merge phase at the end.

We optimize our implementation by using a key observation, that the properties of the hash function remain the same if we count number of *trailing* zeros (NTZ) instead of NLZ. The NTZ operation takes only 4 cycles on a dpCore as compared to 13 cycles for a NLZ due to hardware support for a *popcount* instruction. Instead of a static schedule, we partition the input set into multiple chunks and implement work stealing on the across cores using the ATE hardware atomics. The variable latency multiplier on the dpCores makes this dynamic scheduling essential to avoid long tail latencies. We also use the DMS to read and write buffers at peak bandwidth. We further optimize the x86 version by using atomics for synchronization and SIMD intrinsics. The hash function is at the heart of the HyperLogLog algorithm, and we compare the performance of the DPU for 2 common hash functions, Murmur64 and CRC32. The DPU has hardware acceleration for CRC32, making the CRC implementation almost 9× better than the x86 implementation. The Murmur64 implementation does poorly on the DPU due to the high latency multiplier.

5.5 JSON Parsing

The JavaScript Object Notation (JSON) is an increasingly popular format among many applications that store and analyze large amounts of data. After evaluating open source C/C++ implementations of JSON (JSON11, RAPIDJSON, SAJSON) we selected SAJSON [6] as our best performing, portable baseline. For a benchmark, we populate JSON records with keys corresponding to the TPCH line items table. The data types hence consist of a mixture of integers, strings, dates and populate approximately 1GB of records. For this workload, SAJSON is able to parse the input data at 5.2 GB/s on our x86 machine, achieving an IPC of 3.05. However, on the RAPID DPU, it only achieves a throughput of 645 MB/s. The switch-case anatomy emits a large number of instructions, and lack of hardware branch prediction on the simple dpCores results in a high 13.2 cycles per byte.

Instead of a nested branching structure, we coerce a jump-table by first loading the next byte in the input token stream, and branching conditionally based on the loaded character. Given JSON's relatively small grammar (12 states), the parse table size fits within 23 KB. To allow concurrent processing on all dpCores, the JSON file (in memory) is split into per-core chunks. To further avoid synchronization that would be required if a JSON record straddled the chunk boundary between two dpCores, each dpCore allocates and reads an extra chunk. During parsing, the extra bytes are parsed as the last bytes of the dpCore processing the previous chunk and ignored by the dpCore which encounters them in its first chunk. The efficiency of prefetching buffers using the DMS makes this overhead is negligible. The DMS also triple-buffers the data in 8 KB chunks, with a padding size of 1 KB to avoid the chunk-straddling issue mentioned above. These optimizations allow our DPU implementation to process the above dataset at 1.73 GB/s using 32 dpCores, with an improvement of 8× in terms of performance/watt over SAJSON.

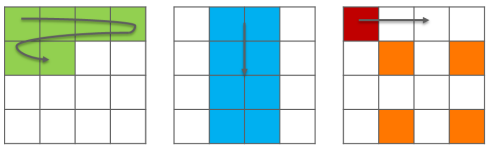

Figure 17: The three types of data access pattern in the disparity computer vision workload.

5.6 Disparity

A Disparity map[42] provides detailed information on the relative distance and depth between objects in the image from two stereo images each taken with slightly different camera angles. This involves computing the pixel-wise difference between the two images by shifting one of the images by X pixels, where X varies from 0 to a given `max_shift` parameter. This well-studied computer vision workload is known to be data intensive [57] whose memory accesses need to be carefully orchestrated to efficiently utilize the available memory bandwidth. The vision kernels in disparity involve three distinct data-access patterns as shown in Figure-17. The most challenging data-access patterns are the columnar and pixelated-pattern. The software-managed DMEM via DMS makes orchestrating these access patterns significantly easier on the DPU. For instance, the pixelated access pattern is reduced to gathering pixels with two different strides into two sections of the DMEM.

In order to efficiently parallelize the disparity computation across multiple cores, we experimented with both fine-grained and coarse-grained parallelization approaches each with different trade-offs. Under the fine-grained approach, we split the input images into distinct chunks or tiles of pixels, one for each dpCore, where they cooperatively compute the disparity kernel in lockstep. This approach requires non-trivial amounts of system-wide barriers for synchronization between the computer vision kernels. On the other hand, the coarse-grained approach splits the work of computing disparity by independently computing disparity for a distinct pixel shift per core with a final result aggregation stage across cores. This reduces the amount of synchronization across cores, but does not efficiently utilize the available memory bandwidth. The low-latency synchronization via ATE and a rich DMS interface enabled an efficient fine-grain parallel disparity implementation with 8.6× better performance/watt relative to an OpenMP-based parallel implementation on x86.

6 RELATED WORK

The Q100 architecture [62] is the closest related work to our system. It consists of a collection of heterogeneous accelerator tiles which are used in conjunction with a coarse-grained instruction set that corresponds to basic SQL operations and processes streams of data. In contrast, our ISA is more general-purpose and fine-grained; we provide generic acceleration features (e.g.: for asynchronous data movement and partitioning) that can be used along with our ISA to accelerate many analytic workloads other than SQL. We have also fabricated our DPU and performed experiments with software running on a real chip as opposed to a simulation.

HARP [61] is fundamentally different from our DMS in several ways. Firstly, its model of operation is different in that it cannot partition the input data stream to all cores and requires a high-functionality partitioning engine for every core. Our system decouples the number of high-functionality engines from the number of cores (thus reducing complexity, cost and power), and can partition the input data to all cores (thus requiring no separate synchronization between cores). Secondly, it requires the core to execute instructions for every 64 bytes of data, keeping the core busy. In contrast, our system uses an asynchronous interface that allows the DMS to partition several KBs of data while the core processes data that has already been partitioned. Thirdly, HARP does not allow the core to immediately use the partitioned data for further processing. Instead, the partitioned data has to be written to DRAM before the cores can load and process it.

DeSC [27] aims to improve the performance of hardware accelerator equipped systems by addressing the memory bottlenecks appearing when traditional cores have to communicate data to the accelerators. CoRAM [19] is a scalable memory hierarchy architecture that connects kernels implemented on reconfigurable fabrics with the external memory interface available in FPGA-based systems. In contrast, our DMS is specifically designed to accelerate data movement and partitioning between DRAM and dpCores while allowing the cores to concurrently process the incoming data.

Using a similar power envelope (250W) and process node (16nm), 21 DPUs are equivalent to a single Tesla P100 GPU [45]. A P100 provides 732 GB/s of peak memory bandwidth, whereas 21 DPUs provide 1612 GB/s of aggregate memory bandwidth. The DPUs also provide > 4TB DDR4 memory capacity, with a similar number of compute cores. Aside from coherence issues and DMS optimizations, there is a direct mapping between multi-core CPU code and DPU code, whereas GPUs require specialized SIMD kernels, making DPUs much easier to program.

Specialized compute engines for different domains have been studied [2, 15, 18, 36, 43, 50, 51, 53] ; while these aim to accelerate operations in specific domains, our aim is to improve performance/watt for a wide range of analytic workloads by identifying and accelerating operations common to all of them.

The IBM Cell Broadband Engine (BE) [33] had similar design choices as our DPU architecture such as low power processing units each with a DMA-controlled local software managed memory. In contrast to the DPU, the Cell's DMA engine is tightly integrated with the processor's pipeline and provides very limited functionality. The Cell BE is also not designed to be a scale out architecture; it does not scale to more than 8 SPUs per SoC.

Wimpy node clusters [5, 31, 38] target energy efficiency and do not provide acceleration for data movement, partitioning or core-to-core communication/synchronization, thus lowering their efficiency for complex analytic workloads.

7 CONCLUSION

We learned several lessons in our efforts to architect and program the DPU efficiently. The DMS is a key enabler of performance in most of our workloads, however, it adds an additional layer of complexity in the software stack. Traditional algorithms do not reason about data movement, however, we find it to be critical for efficient terascale processing. Programming the DPU was more challenging than a commodity x86 core, but easier than the significant algorithm redesign needed for a SIMT based programming model of a GPU, or a sub-word SIMD based CPU. We show efficiency gains of 3× - 15× across a variety of applications on a fabricated 40nm DPU chip. The 16nm shrink of our hardware further boosts efficiency by 2.5×. The DPU architecture focuses on a balanced design between memory and compute bandwidth/watt, and between performance and programmability.

8 ACKNOWLEDGEMENTS

The DPU was the result of a multi-year effort by hundreds of people at Oracle Labs. The authors would like to express their deepest gratitude to the entire DPU team, that we list below.

Aarti Basant, Adam Tate, Adrian Schuepbach, Akhilesh Singhania, Anand Viswanathan, Ananthakiran Kandukuri, Andrea Di Blas, Andy Olsen, Aniket Alshi, Ankur A Arora, Ankur P Patel, Aravind Chandramohan, Arno Prodel, Aron Silverton, Ashraf Ahmed, Ashwin Bindingnavale, Balakrishnan Chandrasekaran, Ben Michelson, Benjamin Schlegel, Bharadwaj Ramanujam, Bill Chandler, Bob Mckee, Brian Gold, Brent D Kelley, Charles Pummill, Cheng X Li, Chris Daniels, Chris Kurker, Christopher Gray, Cory Krug, Craig Mustard, Craig Schelp, Craig Stephen, Dan Fowler, Dan Shamlian, Daniel Joyce, Daniel P Chan, Dave A Brown, Dave Schaefer, Davide Bartolini, Dominic Lin, Doug Good, Egeyar Bagcioglu, Eleni Petraki, Enrique Rendon, Eric Devolder, Eric Holbrook, Eric Neiderer,

Eric Shobe, Erik Schlanger, Farhan Tauheed, Felix Schmidt, Francois Farquet, Gary Cousins, Gaurav Chadha, Gerd Rausch, Gong X Zhang, Harish Gopalakrishnan, Ilknur Cansu Kaynak, Indu Bhagat, J Watkins, Jarod Wen, Jarrod Levulett, Jason M Robbins, Jason Z Wu, Jeff Boyer, Jeff Daugherty, Jeff Kehl, Jen Cheng Huang, Jennifer Sloan, Joe Wright, John Coddington, John Fernando, John Kowtko, Jon Loeliger, Kanti Kiran, Keith Hargrove, Keivan Kian Mehr, Ken Albin, Ken Goss, Kirtikar Kashyap, Lance Hartmann, Leo Lozano, Luai Abou-Emara, Mark Esguerra, Mark J Nicholson, Matthew Wingert, Michael Duller, Michelle Ban, Mike Dibrino, Min Jeong, Mohit Gambhir, Mukesh P Patel, Naveen Anumalla, Negar Koochakzadeh, Nihar Shah, Niranjan D Patil, Nithya Narayanamurthy, Nitin Kunal, Nyles Nettleton, Peter Bradstreet, Peter X Hsu, Phanendra Gunturi, Pit Fender, Rajul Amin, Ram Mantha, Ram Narayan, Rami Zarrouk, Raul M Martinez, Ricardo Marquis, Richard Emberson, Rishabh J Jain, Rita Ousterhout, RJ Rusnak, Robert Wasmuth, Ron Goldman, Ryan Bedwell, Saad Zahid, Sabina Petride, Sasitharan Murugesan, Seema Sundara, Semmal Ganapathy, Senthil Rangaswamy, Shenoda Guirguis, Srivatsan SS Srinivasan, Stratos Papadomanolakis, Thomas Chang, Tom Saeger, Tom Symons, Tony Vu, Venu Busireddy, Victor Skinner, Vijayakrishnan Nagarajan, Vikas A Aggarwal, Vishal Chadha, Vivek Vedula, Walter Esling and Yu Luan.

REFERENCES

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column-oriented Database Systems. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1664–1665. https://doi.org/10.14778/1687553.1687625

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 265–283. http://dl.acm.org/citation.cfm?id=3026877.3026899

[3] Sandeep R. Agrawal, Christopher M. Dee, and Alvin R. Lebeck. 2016. Exploiting Accelerators for Efficient High Dimensional Similarity Search. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/2851141.2851144

[4] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. 2014. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 19–34. https://doi.org/10.1145/2541940.2541956

[5] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. 2009. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/1629575.1629577

[6] Chad Austin. 2013. SAJSON: Single-Allocation JSON Parser. (2013). https://chadaustin.me/2013/01/single-allocation-json-parser

[7] C. Bahlmann, B. Haasdonk, and H. Burkhardt. 2002. Online handwriting recognition with support vector machines - a kernel approach. In *Proceedings Eighth International Workshop on Frontiers in Handwriting Recognition*. 49–54. https://doi.org/10.1109/IWFHR.2002.1030883

[8] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[9] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[10] Luca Becchetti, Carlos Castillo, Debora Donato, Stefano Leonardi, and Ricardo Baeza-Yates. 2006. Using Rank Propagation and Probabilistic Counting for Link-Based Spam Detection. In *In Proceedings of the Workshop on Web Mining and Web Usage Analysis (WebKDD.*

[11] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[12] Christian Böhm, Stefan Berchtold, and Daniel A. Keim. 2001. Searching in High-dimensional Spaces: Index Structures for Improving the Performance of Multimedia Databases. *ACM Comput. Surv.* 33, 3 (Sept. 2001), 322–373. https://doi.org/10.1145/502807.502809

[13] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the Seventh International Conference on World Wide Web 7 (WWW7)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 107–117. http://dl.acm.org/citation.cfm?id=297805.297827

[14] L. J. Cao, S. S. Keerthi, Chong-Jin Ong, J. Q. Zhang, U. Periyathamby, Xiu Ju Fu, and H. P. Lee. 2006. Parallel Sequential Minimal Optimization for the Training of Support Vector Machines. *Trans. Neur. Netw.* 17, 4 (July 2006), 1039–1049. https://doi.org/10.1109/TNN.2006.875989

[15] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783710

[16] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[17] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 609–622. https://doi.org/10.1109/MICRO.2014.58

[18] Eric S. Chung, John D. Davis, and Jaewon Lee. 2013. LINQits: Big Data on Little Clients. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/2485922.2485945

[19] Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An In-fabric Memory Architecture for FPGA-based Computing. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '11)*. ACM, New York, NY, USA, 97–106. https://doi.org/10.1145/1950413.1950435

[20] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, 339–350. http://dl.acm.org/citation.cfm?id=1325851.1325893

[21] William J. Dally. [n. d.]. GPU Computing to Exascale and Beyond. In *Plenary keynote, SC '10*.

[22] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 37–48.

[23] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms (DMTCS Proceedings)*, Philippe Jacquet (Ed.), Vol. AH. Discrete Mathematics and Theoretical Computer Science, Juan les Pins, France, 137–156. https://hal.inria.fr/hal-00406166

[24] Sanjay Ghemawat and Paul Menage. [n. d.]. TCMalloc: Thread-Caching Malloc. ([n. d.]). http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[25] Frédéric Giroire. 2009. Order Statistics and Estimating Cardinalities of Massive Data Sets. *Discrete Appl. Math.* 157, 2 (Jan. 2009), 406–427. https://doi.org/10.1016/j.dam.2008.06.020

[26] G. Grubb, A. Zelinsky, L. Nilsson, and M. Rilbe. 2004. 3D vision sensing for improved pedestrian safety. In *IEEE Intelligent Vehicles Symposium, 2004.* 19–24. https://doi.org/10.1109/IVS.2004.1336349

[27] Tae Jun Ham, Juan L. Aragón, and Margaret Martonosi. 2015. DeSC: Decoupled Supply-compute Communication Management for Heterogeneous Architectures. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 191–203. https://doi.org/10.1145/2830772.2830800

[28] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[29] Tayler H. Hetherington, Mike O'Connor, and Tor M. Aamodt. 2015. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 43–57. https://doi.org/10.1145/2806777.2806836

[30] Tayler H. Hetherington, Timothy G. Rogers, Lisa Hsu, Mike O'Connor, and Tor M. Aamodt. 2012. Characterizing and evaluating a key-value store application on

heterogeneous CPU-GPU systems. In *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '12)*. IEEE Computer Society, Washington, DC, USA, 88–98. https://doi.org/10.1109/ISPASS.2012.6189209

[31] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 314–325. https://doi.org/10.1145/1815961.1816002

[32] Thorsten Joachims. 1998. Text Categorization with Suport Vector Machines: Learning with Many Relevant Features. In *Proceedings of the 10th European Conference on Machine Learning (ECML '98)*. Springer-Verlag, London, UK, UK, 137–142. http://dl.acm.org/citation.cfm?id=645326.649721

[33] James A Kahle, Michael N Day, H Peter Hofstee, Charles R Johns, Theodore R Maeurer, and David Shippy. 2005. Introduction to the Cell multiprocessor. *IBM journal of Research and Development* 49, 4.5 (2005), 589–604.

[34] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. 2013. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 56–65. https://doi.org/10.1109/IISWC.2013.6704670

[35] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (Aug. 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[36] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[37] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural Support for Fine-grained Parallelism on Chip Multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 162–173. https://doi.org/10.1145/1250662.1250683

[38] Willis Lang, Jignesh M. Patel, and Srinath Shankar. 2010. Wimpy Node Clusters: What About Non-wimpy Workloads?. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN '10)*. ACM, New York, NY, USA, 47–55. https://doi.org/10.1145/1869389.1869396

[39] M. Lichman. 2013. UCI Machine Learning Repository. (2013). http://archive.ics.uci.edu/ml

[40] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 36–47. https://doi.org/10.1145/2485922.2485926

[41] Ken Mai, Tim Paaske, Nuwan Jayasena, Ron Ho, William J Dally, and Mark Horowitz. 2000. Smart memories: A modular reconfigurable architecture. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*. IEEE, 161–171.

[42] David Marr and Tomaso Poggio. 1976. Cooperative computation of stereo disparity. In *From the Retina to the Neocortex*. Springer, 239–243.

[43] Rene Mueller and Jens Teubner. 2010. FPGAs: A New Point in the Database Design Space. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10)*. ACM, New York, NY, USA, 721–723. https://doi.org/10.1145/1739041.1739137

[44] Don Murray and James J. Little. 2000. Using Real-Time Stereo Vision for Mobile Robot Navigation. *Auton. Robots* 8, 2 (April 2000), 161–171. https://doi.org/10.1023/A:1008987612352

[45] NVIDIA. [n. d.]. NVIDIA TESLA P100 GPU ACCELERATOR. ([n. d.]). http://images.nvidia.com/content/tesla/pdf/nvidia-tesla-p100-PCIe-datasheet.pdf

[46] Md Mostofa Ali Patwary, Nadathur Rajagopalan Satish, Narayanan Sundaram, Jongsoo Park, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, Sergey G Pudov, Vadim O Pirogov, and Pradeep Dubey. 2015. Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In *International Conference on High Performance Computing*. Springer, 48–57.

[47] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 165–178.

[48] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1493–1508. https://doi.org/10.1145/2723372.2747645

[49] Orestis Polychroniou and Kenneth A. Ross. 2014. A Comprehensive Study of Main-memory Partitioning and Its Application to Large-scale Comparison- and Radix-sort. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 755–766. https://doi.org/10.1145/2588555.2610522

[50] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[51] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. 2013. Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/2485922.2485925

[52] Sridhar Ramaswamy, Pablo Tamayo, Ryan Rifkin, Sayan Mukherjee, Chen-Hsiang Yeang, Michael Angelo, Christine Ladd, Michael Reich, Eva Latulippe, Jill P Mesirov, et al. 2001. Multiclass cancer diagnosis using tumor gene expression signatures. *Proceedings of the National Academy of Sciences* 98, 26 (2001), 15149–15154.

[53] Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. 2012. Accelerating Business Analytics Applications. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/HPCA.2012.6169044

[54] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. 2010. Flexible Architectural Support for Fine-grain Scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 311–322. https://doi.org/10.1145/1736020.1736055

[55] A W M Smeulders, M. Worring, S. Santini, A Gupta, and R. Jain. 2000. Content-based image retrieval at the end of the early years. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 22, 12 (Dec. 2000), 1349–1380. https://doi.org/10.1109/34.895972

[56] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. 2015. Big data: astronomical or genomical? *PLoS Biol* 13, 7 (2015), e1002195.

[57] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. 2009. SD-VBS: The San Diego vision benchmark suite. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 55–64.

[58] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. 2014. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*. IEEE, 488–499.

[59] Yipeng Wang, Ren Wang, Andrew Herdrich, James Tsai, and Yan Solihin. 2016. CAF: Core to Core Communication Acceleration Framework. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 351–362. https://doi.org/10.1145/2967938.2967954

[60] Wikipedia. [n. d.]. Ajax (programming). ([n. d.]). https://en.wikipedia.org/w/index.php?title=Ajax_(programming)&oldid=770489771

[61] Lisa Wu, Raymond J. Barker, Martha A. Kim, and Kenneth A. Ross. 2013. Navigating Big Data with High-throughput, Energy-efficient Data Partitioning. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/2485922.2485944

[62] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[63] Shen Yin and Okyay Kaynak. 2015. Big data for modern industry: challenges and trends [point of view]. *Proc. IEEE* 103, 2 (2015), 143–146.

[64] Wei Yu, Tiebin Liu, Rodolfo Valdez, Marta Gwinn, and Muin J Khoury. 2010. Application of support vector machine modeling for prediction of common diseases: the case of diabetes and pre-diabetes. *BMC Medical Informatics and Decision Making* 10, 1 (2010), 16.