A Many-core Architecture for In-Memory Data Processing

ABSTRACT

We live in an information age, with data and analytics guiding a large portion of our daily decisions. Data is being generated at a tremendous pace from connected cars, connected homes and connected workplaces, and extracting useful knowledge from this data is a quickly becoming an impractical task. Single-threaded performance has become saturated in the last decade, and there is a growing need for custom solutions to keep pace with these workloads in a scalable and efficient manner.

A big portion of the power in analytics workloads involves bringing data to the processing cores, and we aim to optimize that. We present the Database Processing Unit or DPU, a shared memory many-core that is specifically designed for in-memory analytics workloads. The DPU contains a unique Data Movement System (DMS), which provides hardware acceleration for data movement and preprocessing operations. The DPU also provides acceleration for core to core communication via a unique hardware RPC mechanism called the Atomic Transaction Engine or ATE. Comparison of a fabricated DPU chip with a variety of state of the art x86 applications shows a performance/Watt advantage of 3× to 16×.

1. INTRODUCTION

The end of Dennard scaling and dark silicon [9] have saturated the performance of x86 processors, with most new generations devoting larger areas of the die to the integrated GPU [15]. With more companies gathering data for use in analytics and machine learning, the volume and velocity of data is rapidly rising, and massive computation and memory bandwidth is needed to process and analyze this data in a scalable and efficient manner. The slowdown in CPU scaling means that only alternative for traditional server farms is scale out, which is expensive in terms of area and power due to additional uncore overheads.

Custom architectures have traditionally faced resistance in datacenters due to their high programmability and bootstrap cost, however, with Moore's law slowing down, most big vendors are considering application specific hardware as a way to get higher performance at lower power budgets. Intel recently announced their foray into a hybrid FPGA solution [11], Microsoft is exploring FPGAs in their datacenters [21], Google is building custom accelerators for machine learning [16] and all major cloud vendors offer GPUs as part of their offerings.

GPUs are quickly becoming an attractive choice in datacenters for their compute capabilities and corresponding power efficiency [2] [1] [14] [13]. However, they are hard to program due to their SIMT programming model and intolerance to control flow divergence. Their dependence on high bandwidth graphics memory to sustain the large number of on-die cores severely constraints their memory capacity. A single GPU with 300+ GB/s of memory bandwidth still sits on a PCIE 3.0 link, reducing their data load and data movement capabilities, which is essential for high ingest streaming workloads as well as SQL queries involving large to large joins.

We analyzed the performance of complex analytics queries on large data structures, and identified several key areas that can improve efficiency. Firstly, most analytics queries needs lots of joins and group-bys. Secondly, these queries need to be broken down into simple streaming primitives, which can be efficiently partitioned amongst cores. Thirdly, this level of scaleout also means that core to core communication must be fast and power efficient. We present the architecture and runtime capabilities of the Database Processing Unit (DPU), designed for database query processing and complex analytics. The DPU is designed to scale to thousands of nodes, terabytes of memory capacity and terabytes/sec of memory bandwidth at rack scale, focusing on a precise compute to memory bandwidth ratio while minimizing power.

The DPU is a shared memory many-core, the hardware reduces power over a commodity Xeon processor by sacrificing features such as out-of-order, superscalar execution, SIMD units, large multi-tier caches, paging and cache coherence, and instead provides acceleration for common data movement operations. A DPU consists of 32 identical dbCores, which are simple dual issue in-order cores with a with a simple (and low power) cache and a programmer managed scratchpad (Data Memory or DMEM). For an equal power envelope, DPUs provide higher memory capacity and higher aggregate memory bandwidth as well as compared to GPUs, making them a better choice for big data workloads. DPUs achieve this efficiency by recognizing data movement and inter-core communication as key enablers of complex analytics applications, and provide hardware acceleration for these functions.

The DPU programming model is based on scalability and reaching peak memory bandwidth, as each DPU is designed with a balanced compute to memory bandwidth ratio of 2 cycles of compute per byte of data transferred from DRAM. A DPU relies on a programmable DMA engine called the Data Movement System (DMS), that allows it to efficiently stream data from memory. The DMS allows for efficient line speed processing of in-memory data, by allowing data movement and partitioning between DRAM and a dbCore's

scratchpad asynchronously, leaving the dbCore to perform useful work. The DMS can perform many common stream operations like read, write, gather, scatter and hashes at peak bandwidth, allowing for efficienct overlap of compute and data transfer. This also means that dbCores are less reliant on caches, allowing us to dramatically simplify the cache and reduce power consumption. The DMS also communicates with the dbCores via a unique Wait For Event (WFE) mechanism, that allows for interrupt-free notification of data movement, further reducing power and improving performance.

Unlike conventional DMA block engines, the DMS is independently programmable by each dbCore, through the use of specialized instructions called *descriptors* that are populated and stored in the dbCore's scratchpad. Chaining descriptors together to form a pipeline allows for the DMS to be used in a variety of ways. For example, efficient access to large data structures requires hardware support for partitioning operations. [24]. The DMS allows for efficient data partitioning across any/all cores, either based on a range of values, or based on a radix, or on the basis of a CRC32 hash across any number of keys. The results of these partitioning operations are placed in the respective dbCore's scratchpad for further processing.

x86 processors focus on programmability and single thread performance, GPUs focus on compute bandwidth, DPUs focus on programmability and aggregate memory bandwidth/Watt at scale. The DPU is envisioned as part of a larger system, where thousands of DPUs are connected via an Infiniband tree at rack scale, providing > 40000 dbCores, > 10 TB/s of aggregate memory bandwidth and > 1 TB/s of network bandwidth at 20 kW of power. Achieving rack scale gains requires fulfillment of two objectives - Improved algorithms for scalability across these tens of thousands of cores, and improved performance/Watt on a single DPU, which translates into improved performance/Watt of a rack. We taped out and manufactured a DPU using a 40 nm process for benchmarking purposes, and focus our evaluation on this single DPU for this work. Our primary contributions are:

- We present the Data Processing Unit (DPU), a many-core architecture for in memory data processing focused on highly parallel processing of large amounts of data while minimizing power consumption.
- We introduce the Data Movement System (DMS), a DMA engine which provides efficient data movement, partitioning and pre-processing capabilities. The DMS has been uniquely designed to minimize the amount of software needed to control the DMS in critical loops.
- We present an Atomic Transaction Engine (ATE), a mechanism that implements synchronous and asynchronous Remote Procedure Calls from one core to another, some of which are executed in hardware, allowing fast execution of functions without interrupting the remote core.
- We design and fabricate the DPU, and evaluate its performance across a large variety of workloads, achieving 3× to 15× improvement in performance/Watt over a single socket Xeon processor.

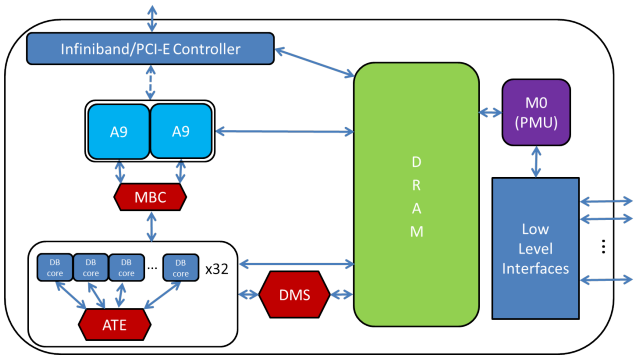


Figure 1: Block diagram of a DPU

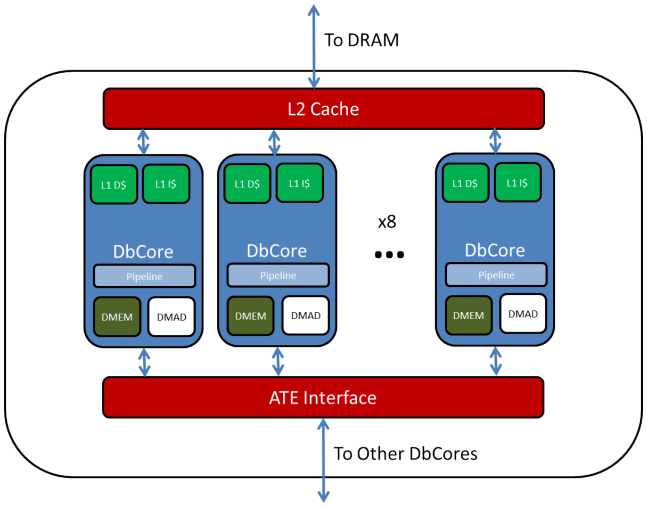


Figure 2: Block diagram of a Macro

- We showcase unique hardware software co-design aspects of our approach towards DPU application performance across a variety of workloads.

Section 2 describe the energy efficient architecture of the DPU and its core components. Section 3 briefly describes the runtime used to simplify application development on the DPU. We then evaluate the performance of our hardware using a few microbenchmarks in Section 4. Finally, we look at the performance and energy efficiency of the DPU across a variety of real world applications in Section 5.

2. DPU ARCHITECTURE

The DPU is a System-On-a-Chip (SoC) that defines a basic processing unit at the edge of a network topology. The DPU is designed with power efficiency and maximizing data parallelism as its principle constraints. Figure 1 shows the block diagram of a DPU. The DPU is designed in a hierarchical manner, 8 identical dbCores form a *macro*, and the DPU contains 4 such identical macros. DbCores in a macro share a L2 cache (Figure 2, and each macro can be power gated independently of other macros. The primary computation in a DPU is carried out on 32 identical dbCores, which can communicate via an ATE interface. Each dbCore reads/writes

to main memory (DRAM) using the DMS, or via a 2 level simplified cache hierarchy. Each DPU has a dual core ARM A9 processor on board as well, which manages the Infiniband networking stack, and provides a high bandwidth interface to other DPUs or a host machine. A Cortex M0 processor serves as a Power Management Unit to control the various power modes of the DPU and provides health monitoring functionality for the board. A Mailbox messaging interface provides a peer to peer communication interface between the dbCores, ARM cores and the M0 processor.

2.1 DbCore Architecture

The DbCore features the Extensible RISC Instructions for Queries, ERIQ, instruction set architecture. The instruction set provides a base instruction set for general purpose computing as well as specialized instruction set to facilitate efficient processing of database queries. Each dbCore is designed to nominally operate at 800MHz, although capable of 1GHz speeds with voltage and frequency scaling.

All instructions in the ISA are 32-bits long and are aligned on a 32-bit boundary. The executable is always loaded in the 0-4GB physical address range in memory to allow addressing with a 32-bit instruction pointer. The ISA consists of 32 64-bit registers with instructions that explicitly operate on either 32-bit quantities or 64- bit quantities. The ISA permits the C language *int* type to be 32-bits to encourage software to operate on 32-bit quantities, while allowing for the pointer type to be 64-bits. Instructions that operate on 32-bit quantities sign-extend the result to 64-bits when saving to the register file.

ALU operations. The dbcore supports common arithmetic operations (addition, subtraction, multiplication, division, modulo), logical operations (shift, rotation), and bitwise (and, or, xor, bit set, clear and population count) operations on word (32-bit), double word (64-bit) or subword (8-, 16-bit) operands. Explicit instructions that operate on 32-bit quantities enable the pipeline implementation to minimize power consumption while meeting computing performance goals. The primary motivation for this 64-bit ISA is memory addressing, specifically the pipeline load/stores addressing far more than 4GB of DRAM when processing database queries. The secondary motivation was ease of handling 64-bit columnar data types. In general, instructions that operate on 32-bit quantities produce a 32-bit result and sign-extend the result to 64- bits when saving to the register file. The motivation for sign-extending 32-bit results stems directly from the load/store instructions. The load/store instructions utilize 64-bits to form an effective addresses. The hardware does not support floating point arithmetic natively.

Comparisons and Branches. Compare instructions operate on word or double word registers and set a single bit in the resultant register. Conditional branches are fused compare-and-branch operations that only operate on 32-bit values. Target address is 16 bits but yields a signed, 18-bit PC-relative range (the lower 2-bits are implied zeros since instructions must be aligned on a 32-bit boundary). Unconditional branches directly identify the branch target via immediate or register operands and can result from explicit branches ('goto' statements), function calls and return statements or while handling exceptions.

Memory operations. In general, all loads read the value from an address in memory and either sign- or zero-extend the value to 64- bits as the value is written into the register file. Two addressing modes are available. The offset addressing mode adds an immediate value to a base register value to form the address. The scaled indexed addressing mode adds two registers together to form the address, the first register being a base and the second register being an index, where the index is scaled by 1, 2, 4 or 8 prior to adding to the base register. All load/store accesses must be naturally aligned. Conditional stores examine the least significant bit of a register to determine whether or not to perform the store.

Address spaces and protection. The address space is broken into physically addressed DRAM regions (up to 8GB DDR memory on RAPIDv1), control, configuration and special purpose register spaces, and a core-local data memory (scratchpad SRAM, called DMEM—32 KB in RAPID-v1). In the absence of a conventional memory management unit (MMU), hardware watchpoint registers allow the loader software to set sentinel boundaries demarcating legal code and data regions. Any dbcore executable overreaching these boundaries causes a hardware exception, thus preventing memory errors and potential vulnerabilities. The DMEM is core-local SRAM mapped at a fixed base address outside the physically addressed DRAM range. Any load or store to the 32 KB address region offset from this base address results in an in-pipeline (cache-hit latency) access to this memory. Software manages the contents of this scratchpad memory with explicit data-movement commands to the DMS. In contrast to the software managed DMEM, each dbcore is also equipped with traditional data and instruction caches (16 KB And 8 KB respectively in RAPID-v1). Hardware populates and evicts these direct-mapped caches on load and store misses to physical memory. In addition a set of 8 dbcores (dbcore macro), share a 256 KB, 4-way associative level-2 cache, which is also implicitly managed by hardware on misses to the L1 instruction and data caches.

Consistency and Coherence. The dbcore does not perform invalidations or updates to remote (i.e., other dbcores) caches when populating or evicting a dbcore's cache (L1 or L2) in response to misses. Instead, the ISA provides explicit operations (*CacheOps*) which can be issued from the dbcore pipeline to invalidate or writeback L1 or L2 cache blocks by address or index. Software uses these CacheOp instructions in conjunction with the remote procedure call mechanism (described below) to ensure coherent memory accesses. All load and store misses cause the pipeline to stall. An explicit *sync* instruction ensures all previous memory operations (including evictions) and serves as a memory fence to ensure program consistency.

Data processing operations. The dbcore supports bit-vector load (BVLD), filter (FILT), and CRC32 hashcode generation as single-cycle instructions to accelerate database operations such as filters, joins and late materialization. The CRC32 instruction computes a seeded 32-bit Castagnoli polynomial which can be used for certain hash-value computations, and for checksums. The BVLD and FILT instructions are designed to filter through sparsely populated columns efficiently in a loop. The program first populates the column into a contiguous buffer in the DMEM address space, notes the

base address in a special register, then repeatedly performs the BVLD And FILT operations to compute offset of the next valid column item, load the element from DMEM, and perform the filter comparison operation. The instructions also enable efficient generation of operations such as population counts and scatter-gather masks.

Event and FIFO processing. The pipeline issues instructions also to the Atomic Transaction Engine (ATE) to perform remote procedure calls, and to the Data Movement System (DMS) to move data between the DMEM and DRAM. The program first constructs descriptors for such operations, then enqueues the descriptor by invoking a *push* instruction identifying the engine (ATE, or DMAC queue as the case may be). To receive completions for the pushed operations, the ISA defines a wait-for-event (WFE) instruction with an event identifier. The above mechanisms allow the dbcore pipeline, and thus the software to: (i) utilize the DMS engines with concurrently pending, programmed memory requests, and (ii) execute operations on a specific dbcore and receive the return value.

Interrupts and Exceptions. The dbcore is designed primarily to execute data processing operations and expect limited interruptions. Specifically, there are three external interrupts which switch execution context.

- When a dbcore pushes an ATE descriptor, the ATE controller raises an interrupt on the target dbcore (which could be the issuing dbcore itself). The ATE interrupt handler causes the pipeline to execute instructions from the remote procedure until it explicitly returns from the interrupt. This feature enables software to construct critical sections and shared data structures.
- When a dbcore or ARM core pushes a mailbox message into a target dbcore's FIFO, the mailbox controller hardware raises an interrupt on the target dbcore. The mailbox interrupt handler minimally enqueues the mailbox FIFO message into a software structure for further processing and returns to the main context. This feature allows software to communicate with external nodes across network fabrics.
- The DPU's System Interrupt Controller (SIC) can be programmed to deliver general purpose I/O interrupts to the dbcore. While most such interrupts are handled by the one of ARM cores in the system, the timer interrupt allows user code running on the DPU to make scheduling and priority decisions.

Exceptions arising from within the dbcore pipeline, such as unaligned data and instruction accesses or divide by zero errors, cause hardware to switch the program counter to specified offsets into an exception vector table. The architecture also implements a special *debug* exception which allows an external debug process to attach to a dbcore executable's instruction stream.

Pipeline execution throughput. The dbcore can issue instructions on two pipelines each cycle. One pipeline issues integer and branch instructions to the ALU, while the other pipeline issues memory operations to the load store unit. Certain instructions (e.g., *sync*) prevent dual issue. Hardware features such as two sets of BVLD/FLIT column base registers are designed to extract full pipeline throughput when coupled with software techniques such as loop unrolling and pipelining. Multiply, divide and modulo instructions stall the ALU pipe for multiple cycles until they complete execution, and bypass their results to downstream instructions. A 12 entry instruction buffer captures tight loops and issue such basic blocks while bypassing the instruction cache. Conditional branches are predicted to be taken when backward, and not taken when forward. Mispredictions cause the pipeline to stall for two cycles. The ATE controller may additionally issue load and store requests to cacheable memory or to the DMEM. No instructions are issued from the instruction cache or loop buffers during these cycles.

2.2 Atomic Transaction Engine (ATE)

Traditional synchronization mechanisms like locks and mutexes are costly and complex to implement and the programmer needs to ensure visibility of the state of the synchronization mechanism throughout the system. We implement an "Atomic Transaction Engine" (ATE) which enables the execution of remote procedure calls (RPC) on other dbCores on the DPU. The ATE guarantees that two RPCs to the same dbCore will be executed atomically, which then allows them to implement mutually exclusive code segments. This also means that all operations to "shared" data must be performed using the ATE, to ensure that data read write ordering is maintained. Atomic RPCs were first explored in the context of distributed systems to ensure consistency in case of node failures [18]. Our use of hardware accelerated atomic RPCs to provide synchronization in a multicore system is novel to the best of our knowledge.

The ATE adds two instructions to the dbCore ISA - RPC No Return (RPCNR) which is used for routines that do not require data to be returned to the calling dbCore, and RPC With Return (RPCWR), we returns data to the calling dbCore. These instructions require 2 operands, a RPC identifier and the address in DMEM for the first payload word. The ATE is programmed with descriptors, which are sent over the ATE network to the other dbCore. The descriptor is a 32 byte structure, and the arguments to the RPC need to be packed into these 32 bytes. If the arguments exceed the space, arguments can be placed in DRAM as well and only a pointer to the arguments will be shipped over the ATE. In this case, due to lack of cache coherence, the runtime system and the application need to ensure visibility of the arguments to the remote dbCore.

When a dbCore issues a RPC instruction, it is routed over the dbCore interconnect to another dbCore, or potentially itself. The ATE on the receiving dbCore places the RPC and its payload in the ATE Receive Queue (reserved in DMEM). The ATE contains a state machine which removes RPCs from the ATE Receive Queue and executes them. The ATE provides hardware support for simple atomic operations like read, write, increment and compare and swap. These low latency routines are executed in hardware, and the remote dbCore is not interrupted. For RPCs which are not executed by hardware, an interrupt is sent to the dbCore and a software interrupt service routine interrogates the RPC Descriptor Register to determine the RPC ID and processes the RPC appropriately.

2.3 Data Movement System (DMS)

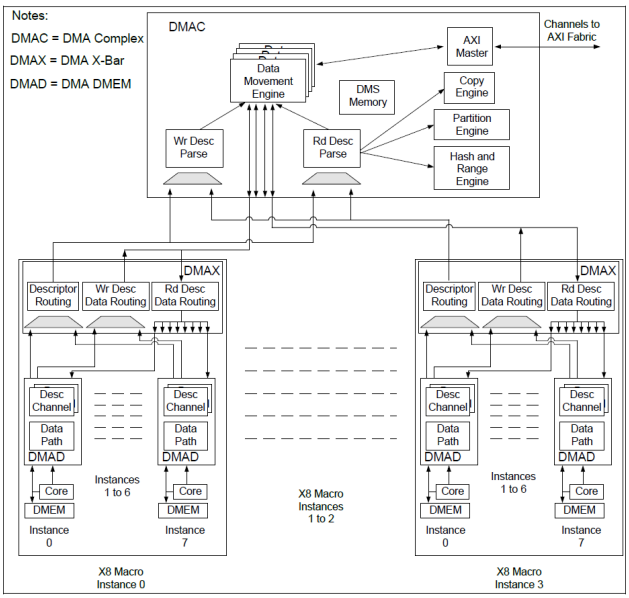


Figure 3: Data Movement System (DMS)

A majority of database operations are focused more on data movement rather than on computation. To support these operations, we have implemented an intelligent DMA engine, termed the Data Movement System, or DMS. Unlike conventional block DMA engines, the DMS understands database data formats and can perform a variety of functions, including data partitioning and projection, hash computation, and array transposing (row-major to column-major or vice-versa), all while transfering data. The DMS is fully programmable by software using linked descriptors in the dbCore's DMEM, enabling software to combine the capabilities of the DMS in a flexible manner to suit various processing needs.

Figure 3 shows the overall architecture of the DMS, a data highway built from three separate units. The central DMA Controller or DMAC performs all tasks that are common to all 32 dbCores such as descriptor parsing and is responsible for the bulk of the data movement in the DMS. It comprises most of the control, buffering and data path processing of DMS, allowing most of the DMS logic to be shared across dbCores. The DMA DMEM unit (DMAD) performs the tasks that are unique to each dbCore. It is designed to be as simple as possible and is duplicated on all 32 dbCores. The third unit is the DMA Crossbar (DMAX) which serves as a switch or crossbar in each macro. It acts as a bridge between the eight DMADs to the DMAC.

The DMS provides efficient data movement between the main memory (DDR) and scratch pad memory (DMEM) in the dbCores. While moving data, the DMS can also efficiently perform some preprocessing operations such as scatter/gather, computing hashes and partitioning the data. The DMS provides a robust interface to minimize the software overhead involved in programming it, through the use of descriptors. Descriptor based DMA has previously been used to program on-chip NICs [4], however, our use of the DMS to efficient move data from DRAM to the dbCores for is unique. The DMS is also fully aware of database tuples (multiple columns), and supports multiple data types. Multiple descriptors can be queued up in a hardware managed linked list for processing. A special *loop* descriptor is used to reuse these programmed descriptors multiple times. These descriptors provide ways to specify auto-incrementing the effective address, allowing large columns of data to be moved to DMEM without reprogramming the DMS. These unqiue aspects of the DMS dramatically simplify the programming interface, improving efficiency for operations involving small buffers, where the programming cost cannot be amortized.

For efficient communication with the dbCores, the DMS software interface uses the Wait For Event (WFE) mechanism, allowing buffer flow control without the overhead and latency of interrupts. The DMS interface exposes 31 DMS events per dbCore that software can use to communicate with the DMS hardware. DMS descriptors have a notify event field that specify the event to be asserted when the operation is completed. They also have a Wait event field that specifies which event the descriptor to be waited on. Software can set or clear the specific events directly. Software can also wait on a specific event using WFE instruction. It should be noted that while the dbCore is waiting for an event using the WFE instruction, it is automatically put in to a low power state where most clocks are gated. The WFE instruction can be thought of as hardware managed low power polling.

Software can use two channels per DMAD (read and write) to submit the descriptors to DMS. DMS descriptors are 16 bytes long, which encodes the source address, destination address and other control information such as auto-increment, event mangement in a compact way. Software programs the DMS by pushing the address of the descriptor to the DMAD's active list, using the FIFO interface on one of the channel. DMAD walks the active list and processes the descriptors in order. Before it executes the descriptor, it checks descriptor from the DMEM and executes the descriptor once the other conditions such as synchronization, events readiness are satisfied. The descriptor information is passed up to DMAC through DMAX. The DMAC executes the descriptor by performing the operations such as moving data from the main memory to DMEM or partitioning the data etc., Upon completion of the operation, it returns the descriptor to DMAD through DMAX. Once the completion notification is reached DMAD, DMAD retires the descriptor. When it retires, it also notifies the core if it is programmed to raise an event.

Table 1 shows the various type of DMS descriptors available to program the DMS. The data movement descriptors are used to program the DMS to move the data betweeen the main memory, core's DMEM, and DMS internal memory. DMS internal memory are buffer space provided by DMS to store column data during data operations. For example, it provides the space for key columns while DMS computes the CRC32 and provides space for columns while data is partitioned among the cores using the hash or range. Loop descriptors specifies the looping operations on the descriptors. AUX descriptors provide a way to provide additional information needed to process other descriptor. This provides a way to extend the descriptors with backward compatibility. Subsection 3.3 describes how the DMS is programmed and presents an example.

Descriptor Type	Main Purpose
DDR->DMEM	Moving data from the main memory to DMEM
DDR->DMS	Moving data from the main memory to DMS internal memory
DMEM->DDR	Moving data from DMEM to the main memory
DMEM->DMS	Moving data from DMEM to the DMS internal memory
DMS->DMEM	Moving data from DMS internal memory to DMEM
DMS->DDR	Moving data from DMS internal memory to the main memory
DMS->DMS	Moving data between the DMS internal memory
Loop	Specifies looping operations
Event	Provides a way for DMS to wait on events and clear events
HARE	Controls HASH and range functions
AUX	Used to provide any additional information needed to process other descriptors
Program	Configures DMS control registers

Table 1: DMS descriptor types

2.4 Mailbox Messaging

The Mailbox Controller (MBC) is used for managing communication between the dbCores, the A9 cores and the M0 processor. It maintains a total of 34 mailboxes, one for every dbCore, one for both A9 cores and one for the M0. The controller is located on the AXI bus, and it is accessible via a standard AXI slave interface by all cores.

For each mailbox the MBC maintains a set of memory mapped data (wr/rd) and control (wr/rd) registers that are used to write (send) or read (receive) messages to and from the corresponding mailbox, along with the corresponding logic that implements the send and receive protocol. An SRAM memory module, which is shared among all mailboxes, is used to store all messages delivered but not consumed yet.

A source core that wants to send a message to a destination core, first acquires exclusive access to the destination mailbox, by accessing atomically the destination WR control register. Once given exclusive access, it checks for available free space and then sends the message, by performing consecutive writes to the WR data register. Finally, it marks send completion by updating the WR control register. The MBC logic updates the counter of received messages for the corresponding mailbox, causing an interrupt to be delivered to the destination core.

Following similar steps, the destination core, upon a mailbox interrupt, will read the RD control register to see how many messages are available. It will then read the data of each message, by performing consecutive reads from the RD data register. The first word of a message is expected to contain the number of words each message consists of, in order for the destination core to identify message boundaries. After the destination core has read all individual words, it updates the RD control register to mark the consumption of the message. At that time, the message counter for this mailbox is decremented and if there are no other messages available the interrupt signal is de-asserted.

The messages exchanged over MBC are always a multiple of 4 bytes, and their size varies between one word and one mailbox's capacity (128 words). The goal of the MBC is to facilitate quick exchange of light-weight messages (i.e., sending a pointer to buffer in memory), while the bulk of the data are communicated through main memory. The network communication among the nodes and the RDBMS is based on this property. For example, a node will place the results of the query execution in a buffer in memory and send a mailbox message to A9 with a pointer to the buffer. Upon receiving the message, the A9 will pass the pointer to the network stack, which will then forward the buffer to its destination.

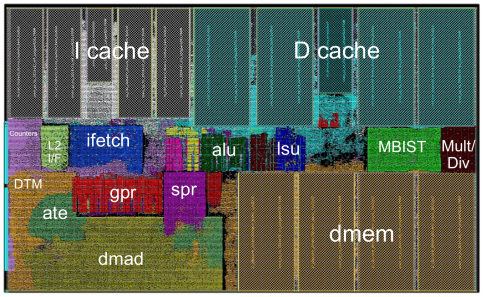


Figure 4: dbCore processor implementation

Overall, the latency of a mailbox message may not be as low as the latency of a more integrated network (i.e., ATE). However, we observed the MBC message throughput to be sufficient to sustain a network message rate that keeps the dbCores utilized, without introducing any communication bottlenecks to the system.

2.5 Fabrication

We fabricated the DPU using a 40 nm process, with a silicon area of 90.63 mm2 and 540 millions transistors, of which 268 million transistors are used for memory cells. Figure 4 shows the implementation of a single dbCore. 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. The DMS proved to be one of the most challenging blocks to verify, due to deep corners in design. We also created a detailed virtual

prototype in software for verification, allowing us to perform end-to-end black box testing of the DMS.

Any hardware is only as good as the software that runs on top of it. We next look at the our runtime. Creating a runtime for the DPU was based on the same principles as its hardware, maintaining a balance between efficiency and programmer productivity. and how we

3. THE DPU RUNTIME

The low power design of a DPU delegates a majority of scheduling and memory management decisions to software. The DPU runtime is guided by same principles, providing a intuitive and powerful API to the programmer, while maximizing optimization potential through the use of the DMS and the ATE messaging interfaces. The guiding principle behind the DPU runtime is energy efficiency by fully exploiting all levels of compute and memory parallelism, and our focus on achieving peak memory bandwidth. Programming the DPU essentially involves asking the following questions:

- How can I partition my data across all dbcores to maximize compute parallelism?
- How can I layout my data in DMEM so I can concurrently load, compute and store data?
- How can I layout my data in main memory to allow the DMS to fetch and store contigous buffers, and achieve maximum memory bandwidth?

3.1 DbCore Programming

The dbcore comes out of reset in a bare state, and the runtime is responsible for reserving the stack and heap in DRAM. It also reserves some space in DMEM for the ATE message queues and runtime state. After initializing the interrupt handlers, control is transferred to the *main* routine. A dbCore is similar in architecture to an embedded MIPS core, allowing for rapid prototyping. We provide standard C library routines such as *printf*, memory and string APIs using the newlib library [23]. Subsequent optimization relies on moving data from the caches into DMEM, and overlapping compute with memory transfer using the DMS.

The programmer must be aware of incoherent caches, and we use the ATE to simplify core to core communication. The DPU allows executing a function pointer on a remote core. This mimics remote procedure calls where an API is called which executes on a remote endpoint as well. This mechanism can be employed to build data structures to which accesses must be performed in a mutually exclusive fashion. We define a serialized interface to allows execution of software RPCs on remote cores.

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

A target core id as well as a function pointer which will be executed on the remote core are specified. In addition to that, the arguments supplied to the RPC function pointer on the remote core are supplied as parameters. To achieve coherency in this 1 to 1 communication style, a visitor specifying the coherency contract for the arguments struct as well as another visitor for the returned data structure are supplied.

The non-coherent caches require that the programming model enforces the programmer to specify how shared structures should be made coherent by the underlying layers. The vistor has a *memory operation* argument that defines which operation should be executed and will be filled in by the runtime. This argument modifies the visitor to either *flush* the data structure on the calling core, and *invalidate* the data structure on the remote core ensuring coherence is maintained. The visitor essentially defines a coherence contract, which can be used by the runtime to perform the necessary actions.

3.2 Scheduling

Each dbCore maintains an independent thread of execution, and a list of pending tasks that are executed in order. The scheduler is designed to be lightweight and runs as a part of this main event loop, and puts the dbCore in a sleep state if the task queue is empty using a *wfe* instruction. We create a lightweight messaging API on top of ATE Software RPCs, and an ATE interrupt on a dbCore adds a task to the local task queue, and wakes up the dbCore. The scheduler resumes execution, and processes all tasks in the task queue. The scheduler relies on the programmer to yield control at the end of a task.

We create a lightweight barrier by using the ATE hardware RPCs. A master core (the core where the barrier is created) acts as the coordinator, and initializes a local variable *count* in its DMEM to zero. Other cores increment this variable by using the atomic add with return ATE RPC, and once *count* reaches the total number of cores, all cores leave the barrier. Cores can check the value of this shared variable via a poll based mechanism using the ATE, however that would create significant contention on the coordinator core. We create a scalable barrier, where each core uses a local wait variable, and spins on this variable after atomically incrementing *count*. Once all cores enter the barrier, the coordinator notifies all the other cores by modifying the local wait variable using an ATE atomic write operation. This would cause the coordinator to perform 31 sequential writes once all cores enter the barrier. We further optimize this based on our architecture, and the coordinator notifies *macro masters*, which in turn notify the cores within their macro, creating a scalable tree like barrier. This improves the time taken for the last core to leave the barrier from 30 us for the sequential case to 12 us for the tree barrier.

3.3 DMS Programming

The DMS in the DPU is fully programmable with the DMS Descriptors as mentioned in the Subsection 2.3. DMS descriptors are like macro instructions or commands which instructs the DMS to move data between the main memory, DMEM, and DMS internal memory, to partition the data, to perform looping operation and to deliver events when data movement is complete.

Listing 1 illustrates how the DMS is programmed for a simple DMA access with double buffering. In this example, the DMS is programmed such that the program consumes 4 million rows of a 4 byte column. First we setup two DDR->DMEM descriptors is setup to read 256 rows with same

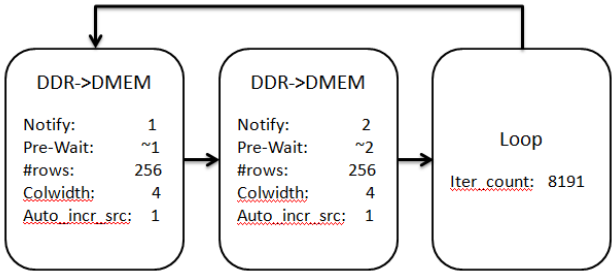


Figure 5: DMS Descriptors

source address and destination address, but with different events. Even though the same destination address is used, the effective address is different due to the auto-increment feature in the DMS. Second, we setup the loop descriptor such that it loops to the first descriptor for 8191 times such that it reads all 4 million rows. Finally, we push the descriptors to the DMS and thereby we trigger the data transfer to DMEM. Figure 5 shows the state of the descriptors in the active list after they are pushed. Once the data is fully transfered to DMEM for the first descriptor (desc0), it will notify the core by raising the event1. The DBCore waits for event1 to be raised and then start consuming the rows. Meanwhile DMS start transfering the data corresponding to the second descriptor (desc1) to DMEM without waiting for dbcore to finish consuming the data. After finish transfering the data, DMS will loop back to first descriptor and wait for the event1 to be cleared. After consuming the data for the first descriptor (desc0), dbcore clears the event, signaling the DMS that the first buffer is free. The overlapping of computation (*consume_rows*()) and the transfering of data with double buffering enables the DPU to process data with high throughput.

```
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, event1);
dms_descriptor* loop = dms_setup_loop(desc0, 511);
dms_push(desc0);
dms_push(desc1);
dms_push(loop);
count = 1;
buffer_index = 1;
events[] = {event1, event2};
do {
  dms_wfe(events[buffer_index]);
  consume_rows();
  clear_event(events[buffer_index]);
  buffer_index = 1 - buffer_index; // toggle buffer
       index
} while (++count != 8192);
```

Listing 1: DMS Programming Example

3.4 Memory Management

The memory available in the system consists of the main memory, which is shared among dbCores and DMS, and the DMEM scratch space locally available to every dbCore. Each one of these two spaces are managed in different way.

3.4.1 Heap Space Allocator

We keep in DRAM instructions (i.e., dbCores' binary image) and data, such as stack space for each dbCore, heap space and global data. The lack of an OS or of a virtual memory management scheme, has led to having all these regions fixed in the memory address space, and share their location with the compiler (i.e., position of the stack) or expose them to the dbCore programmer, either directly or through an API.

The largest fixed memory region is the heap space. We implemented a two-level dynamic memory allocator, that overlooks the heap space and provides dynamic buffer management. The lower level implements a fast buffer allocation/free core-local path that handles the majority of the requests. When a core-local allocator runs out of buffers, it requests for additional memory from the global allocator, which may choose to reply with either a set of buffers of the requested size or a chunk of memory, which the core-local allocator will use to produce buffers of the requested size. Similarly, if the number of free buffers in a core-local allocator exceeds a pre-defined parameter, the core-local allocator will return a number of buffers to the global allocator. Each core can reach the global allocator by performing an ATE request to a specific core, which is tasked with running the global allocator. The code for the global allocator, is always executed within the context of an ATE interrupt, and thus is designed to be as short as possible.

Internally, both allocators are organized as a set of segregated free-lists. The core-local one maintains buckets of sizes ranging from 64 bytes to a few MBs, forwarding any request for larger buffers directly to the global allocator. The global allocator maintains the same set of free lists, to collect returned buffers from the cores and an additional linked list for all the remaining buffer sizes.

The memory allocator participates in the system-wide effort of maintaining (software) cache coherence. At any given time a buffer can be in one of the following three states:

- It can be in the global allocator, which means no part of such a buffer can be found in any part of the memory hierarchy, other than the main memory, and it is not going to be accessed by anyone.
- If in the local allocator and marked as *cache-only*, it can be accessed only through ld/st instructions executed by dbCores, and parts of the buffer can be in different locations in the memory hierarchy (i.e., different caches). It is the programmers responsibility to further enforce which core is the owner of the buffer at any given time.
- If in the local allocator and marked as *DMS-only*, it can be accessed only through DMS operations, and cannot be loaded on any cache.

The state of any buffer can change only when the buffer is free, and either moves to/from the global allocator or it is in the local allocator and the allocator chooses to transition the buffer based on requests and availability. All state transitions require the address range of the buffer to be invalidated, apart from the ones moving from *DMS-only* to any other state.

3.4.2 DMEM Memory Management with a Stack Allocator

The DMEM holds data input to operators as well as intermediate results, shared across operators. The lifetime of this data is short, as the chain of operators forming a task, is usually short and the result of a task is copied back to DRAM. To avoid the overhead of dynamically managing the available DMEM space through a malloc-like API, the operators runtime uses DMEM in a stack-based fashion. Every new operator appends its result to the end of the used DMEM space. At the end of the last operator, the whole stack space is released and made available for the next task.

4. MICROBENCHMARKS

We fabricated a test DPU using a 40 nm process, with 8 GB of DDR3 memory. We first evaluate the performance of the DMS and the dbCores for several small test cases, and ensure that functionality on the DPU works as designed, and measure peak performance of the DMS and ATE across several use cases.

4.1 Read and Write

We first take a look at the Read and Write performance of the DMS. Each dbcore reads 4096 rows from DRAM into DMEM using the DMS, and write the same rows from DMEM to DRAM. Each dbcore reads/writes a buffer of multiple rows at a time containing all columns. We vary the number of columns from 2 to 32 and width of each column from 1 byte to 4 bytes. The data is stored in column major format. Figure 6(a) shows the read bandwidth achieved across all dbcores across different buffer sizes and column widths. There is a small decrease in bandwidth as the number of columns increase. The DMS fetches data one column at a time, and more columns mean more non-contigous DRAM pages are opened, which incurs a small latency for each buffer. Larger buffer sizes amortize fixed DMS overheads achieve higher bandwidths. Using the DMS, the DPU achieves a bandwidth of > 9 GB/s for a buffer size of 8 KB(128 rows/buffer, 4 columns, 4B column widths) which is almost 75% of the peak DDR3 bandwidth, and this is bandwidth we observe in many real applications as well.

4.2 Gather

We study a basic bitvector gather operation, where we gather rows from DRAM corresponding to 1s in a given bitvector into DMEM. Gather is fundamental to the filter operator, wherein we gather rows from DRAM corresponding to ones in a given bitvector. and . We test the DMS against a dense (0xF7) and a sparse (0x13) bitvector. The DMS is designed to perform gather at line speeds, however, due to a RTL bug, the first version of our chip could not utilize the DMS to its full potential(Figure 7). In brief, when all 32 cores issue gather operations, a FIFO that hold bitvector counts in the DMAC overflows and starts dropping counts. DMAD channels in the corresponding dbCores that are waiting for these counts get stuck, causing some dbCores to hang. We create a software workaround for this issue by serializing descriptors across dbCores, ensuring only a single dbCore issues a gather operation at a time, causing low gather bandwidths(Figure 7).

4.3 Hardware Partitioning

Partitioning is critical for achieving high efficiency for data processing operations and we study the efficiency of the partitioning engine in the DMS in terms of its achievable memory bandwidth. The input to the microbenchmark is a relation with four 4 byte columns. The relation is stored in column major format. The microbenchmark uses the DMS's partition engine to partition the input relation into 32-way partitions. First, the DMS is programmed to move the data from the main memory into the DMS internal memory. Once the data is in the DMS internal memory, DMS then triggers the hash or range engine to compute the index of the output partition. Finally, DMS moves the payload column data from the DMS internal memory and sends them to output partitions which are in a core's DMEM. We pipeline these three stages since that DMS uses different resources in each stages. In addition, we used multiple cores to submit the DMS descriptors to reduce the DMS programming overhead per core and used all four available memory controller to maximize the input bandwidth to the hardware partitioning engine.

Figure 8 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. Hash partitioning first uses the hash engine to compute the CRC32 from one or multiple key columns and then uses the CRC32 bits to partition the data. In all the partitioning scheme, the DMS achieves 9.3 GB/s and outperforms the previous published state of art hardware accelerator for partitioning [24], where the partitioning throughput is only 3.13 GB/s. The DMS achieves high efficiency by pipelining the partitioning operations into three stages, by using efficient hash and range engine desigh, which runs at 800 MHz, and the usage of the low latency scratchpad memory to process the partitions.

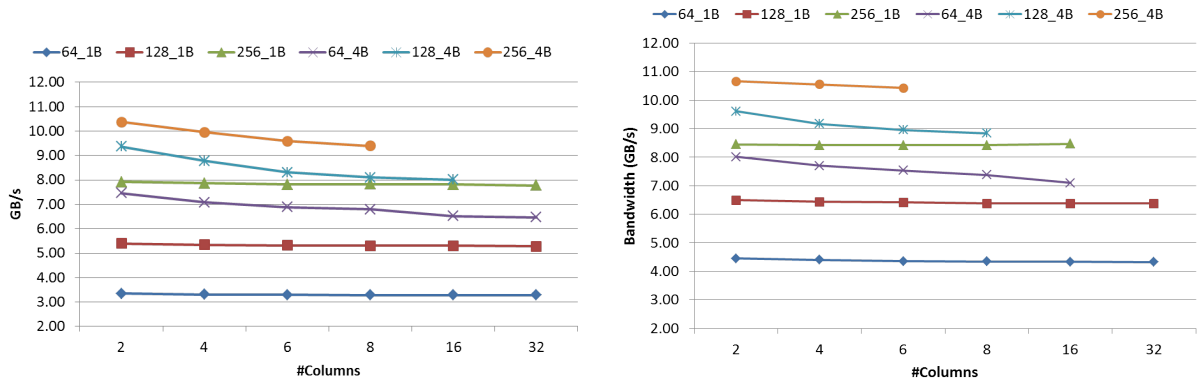
4.4 Atomic Transcation Engine

The ATE is designed for fast core to core communication, and allows dbCores to perform atomic operations on shared data in DMEM or DRAM. We measure the latency of different ATE operations and different payload sizes (Figure 9. Looking at read, an atomic read from a remote dbCore incurs a latency of 80 ns for an 8B value. Writes are faster, as the local dbCore does not need to wait for a return value. Similarly, an atomic addition that returns a value to the caller dbCore takes 100 ns. Software RPCs are much slower due to interrupt overhead, incurring almost 400 ns to read an 8B value.

4.5 Mailbox

The mailbox controller is responsible for the communication between the dbCores and the A9 cores. We evaluated its peak performance with the following traffic patterns:

- A9 to dbCore0, back to back messages of variable size.
- dbCore0 to A9, back to back messages of variable size.
- A9 to dbCores, back to back messages on both directions. dbCore0 is the recipient of A9 messages and dbCore1 the sender. A9 alternates between sending and consuming a message.



(a) Columnwidth = 1B

(b) Columnwidth = 4B

Figure 6: Bandwidth achieved across 32 dbcores for reading (a) and reading and writing(b) data via the DMS. Each line shows the bandwidth achieved for given number of rows per buffer and column width

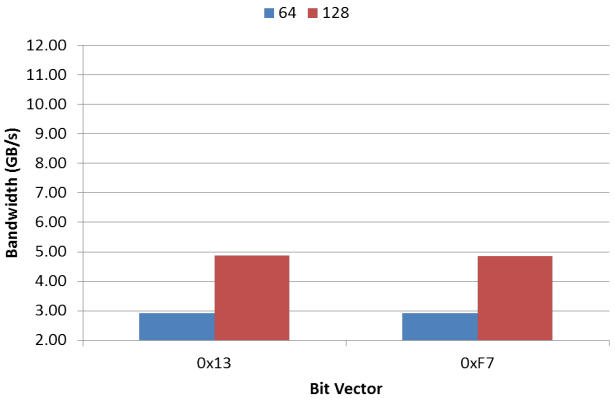


Figure 7: Bandwidth achieved across 32 dbcores for the bitvector gather operation using the DMS. The x-axis

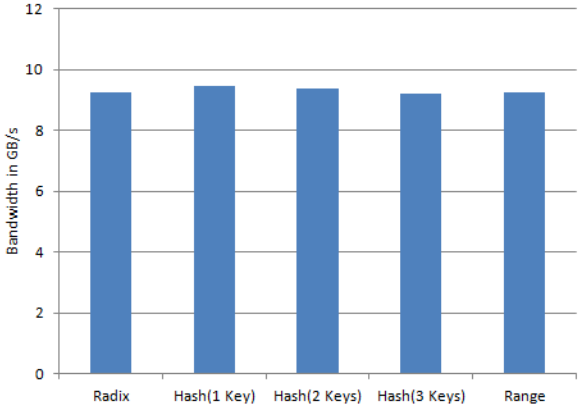


Figure 8: Bandwidth achieved with DMS partitioning engine

- round-trip messages of minimum size, exchanged between A9 and dbCore0.

Figure 10 shows the message throughput rate in thousands of messages per second, as a function of the message payload size (in 4-byte words), for the three traffic patterns that exchange back-to-back messages. The message processing rate at the two ends of the message exchange is different, with A9 having to involve the kernel interrupt to receive the messages and deliver them to the user space of the destination process. On the other side, when a dbCore is receiving a message, the processing path is much shorter. As a result, there is higher throughput when A9 sends messages to dbCores than when receiving. The combination of the two patterns, results naturally in lower throughput, as the A9 has more work to do compared to either previous scenario.

Finally, we evaluated the round-trip throughput of the mailbox communication channel, by having A9 and dbcore) exchange the same message multiple times. We measured the average message rate under that traffic pattern to be 64.184 round trips per second.

4.6 SQL Operations

4.6.1 Filter

The filter operation is a basic database operation (SQL WHERE clause), used to identify elements of a column that satisfy a given condition. We program the DMS to fetch a single column of data (columnwidth = 1B), and vary the number of rows from 512 to 16384. The DMS fetches a tile of the requested size into DMEM, and the dbCores generate a bitvector in DMEM corresponding to the $\leq, <, \geq, >, =$ comparisons with a constant. We accelerate these operations by using the *bvld* and *filt* instructions in the dbCore ISA, and pipeline these operations together with the DMEM read. A single dbCore achieves a bandwidth of 482 millions tuples/second (Figure 11), which translates to 1.65 cycles/tuple. We can see a saturation in dbCore performance as the number of rows are increased, as fetching larger buffers causes the DMS to approach peak memory bandwidth. For 16K rows/dbCore, this allows us to achieve a peak memory bandwidth of almost 9.6 GB/s for 32 dbCores.

4.6.2 Binary Expressions

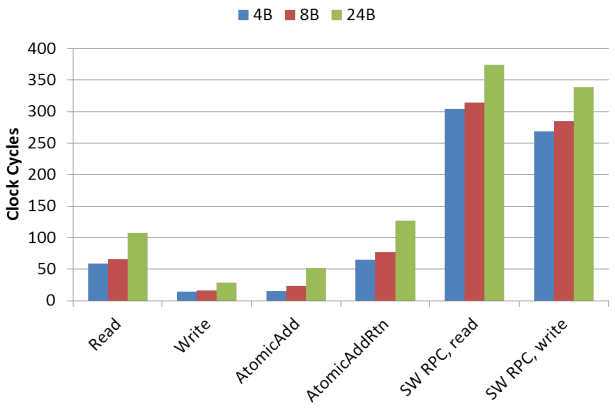


Figure 9: Performance of ATE SW and HW RPCs for different payload sizes

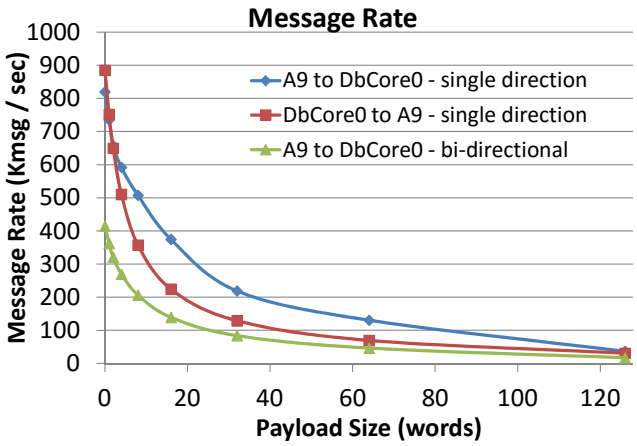


Figure 10: Mailbox throughput as a function of payload size

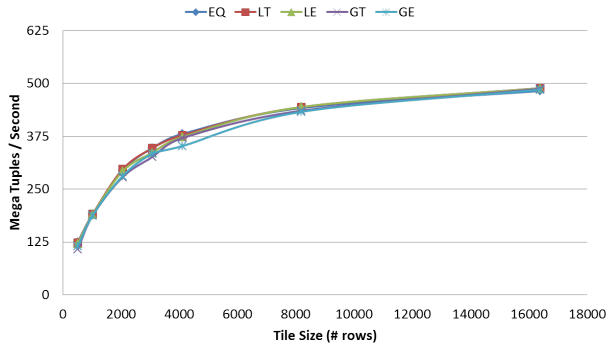


Figure 11: Performance achieved for a single dbCore for the filter primitive

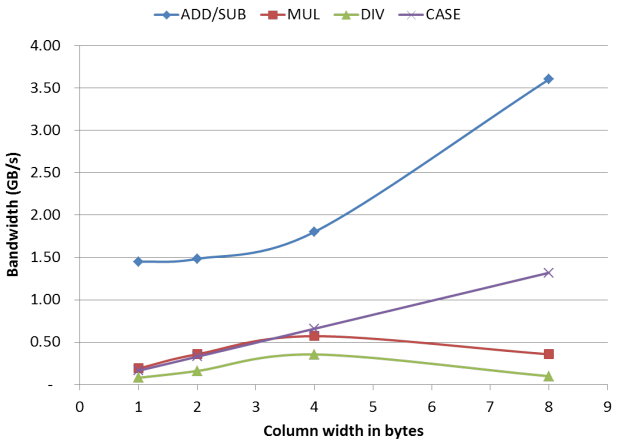


Figure 12: Performance achieved for a single dbCore for the filter primitive

We evaluate binary expressions on the DPU, which are defined as operations on columns of the form ColA EXPR ColB, where ColA and ColB are two database columns, and EXPR is one of ADD, SUB, MUL or DIV. We also evaluate the CASE expression defined as ColA $>$ Const1 then ColB else Const2. We fetch 1024 rows per column using the DMS into DMEM, and measure the achieved bandwidth for different columns widths (Figure 12). The bandwidth for the ADD/SUB expression increases with larger column widths, due to larger buffer being transferred by the DMS. MUL and DIV achieve their peak bandwidth at a column width of 4 bytes, and become compute bound for higher column widths. The CASE expression bandwidth increases linearly with increasing column widths, due to improved DMS bandwidth. CASE achieves lower bandwidth than ADD/SUB due to higher branch misprediction penalties.

5. APPLICATIONS: OPTIMIZING FOR THE 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 that are a good match for our hardware. We find state of the art algorithms from research for these applications, and implement them both on x86 and the DPU. In most cases, similar code runs on both architectures, with threads on x86 translating to dbCores, and cache-friendly memory operations are converted to DMS operations. The DMEM and LLC on the DPU are much smaller than Xeon processors, and we perform additional optimizations to allow working sets to fit in DMEM.

We compare our numbers to a Xeon server, with two Intel Xeon E5-2699 v3 18C/36T processors and 256GB (16 * 16GB Samsung 16GB DDR4 2133MHz) DRAM running at 1600 MHz. For our DPU experiments, all datasets were converted to 10.22 fixed point. There has been a increasing amount of research on using fixed point for machine learning algorithms [12] [7], and we observed no losses in accuracy while comparing with a floating point implementation. A

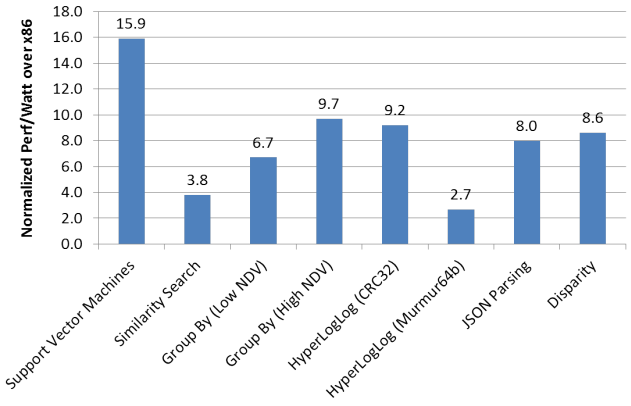


Figure 13: Power efficiency gains of the DPU for several applications

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. We create optimized routines for fixed point multiply, divide, dot product, and exponential functions. For the dot product, we use hand-coded assembly to make sure the element-wise loop fits into the loop buffer, making the backwards branch free. To compute performance/Watt, we assume a TDP of 145 W for the Xeon, and 6 W for the DPU. Figure 13 shows the performance/Watt advantages of the DPU for several applications that we looked at normalized to a corresponding optimized x86 implementation.

5.1 Support Vector Machines

Support Vector machines (SVMs) are widely used for classification problems in healthcare (cancer/diabetes), document classification and handwriting recognition. We implement a variation of the Parallel SMO algorithm proposed by Cao et. al [5] 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 dbCores, 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 dbCores using the ATE as well.

Sending large data values across the ATE is expensive, so we store all data structures in shared DRAM structures, and use a soft coherence protocol to maintain integrity. The sending core flushes the data structure from its cache and notifies the receiving core via an ATE message. The receiving core then invalidates the data structure address range from its cache. 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 [6] implementation on x86. We use 128K samples from the HIGGS [17] dataset for evaluation. Optimal parameters are chosen for LIBSVM (100MB kernel cache, 18 OpenMPI 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, further making the case for fixed point.

5.2 Similarity Search on Text

Text processing is an ubiquitous workload, we use a simple example of similarity search on text to demonstrate the applicability of DPUs to text analytics. The similarity search problem essentially involves computing similarities between a group of queries and a group of documents indexed using the *tf-idf* scoring technique, and coming up with a set of *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) [1]. We leverage recent research on optimizing SpMM on the CPU [20] and the GPU [1] and implement these algorithms on x86 and the DPU. Each query independently searches across the index, making the problem easily parallelizable across multiple threads/dbCores. We search across 4M pages in the English Wikipedia, using page titles as queries, similar to [1].

A SpMM operation between 2 sparse matrices A and B creating C relies on a simple principle, accumulate rows of B corresponding to non-zero columns of A into C. The algorithm relies on using a dense intermediate format for rows of C, which makes indexing easier during accumulation. We range-partition rows of B and C into smaller tiles, allowing this intermediate dense array to fit in the LLC. These tiles are stored in the Compressed Sparse Row (CSR) format, which means we cannot know when a tile ends without actually reading the tile. These irregular access patterns make SpMM a challenging problem for DMS access. Getting good bandwidth and efficiency from the DMS entails fetching large fixed length contiguous buffers. 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 dbcores. 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. This allows us to consume all data in DMEM, increasing bandwidth utilization.

Careful DMEM management improves the effective bandwidth to 5.24 GB/s on the DPU and a 3.9× improvement in performance/Watt over a optimized 256 thread Xeon implementation (effective bandwidth across 36 cores - 34.5 GB/s).

5.3 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) of a given list of expressions within each group. It can be efficiently processed using a hash table as long as the number of distinct groups is small enough [8]. 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. On conventional architectures like x86, query

processing systems rely on large multi-tier caches to hide the latency of accessing a large hash table; even on such systems there is no guarantee that a given section of the hash table will be cache-resident and performance drops significantly when the hash table grows beyond a certain size.

Our query processing software is architected 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; the algorithm initially allocates a minimum amount of space for each structure, and then allocates the rest of the space incrementally using a greedy approach based on the performance gains of allocating extra space to each structure. Typically, input/output buffers don't benefit much from more than 0.5 KB (since the DMS can nearly saturate memory bandwidth at that point) and hence a large part 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. Based on the number of columns involved, we can calculate maximum number of software partitions that can be achieved in one "round" (round-trip through DRAM âĂŞ reading data in and writing it out as separate partitions) at a rate that is close to memory bandwidth; this is because software partitioning internally uses DMEM buffers for each partition. 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 groupby for two cases: low number of distinct values (Low-NDV) and high number of distinct values (High-NDV). In the Low-NDV case, both platforms are able to process the operation at a rate close to memory bandwidth; so the improvement (6.7x) 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.

5.4 HyperLogLog

The HyperLogLog algorithm [10] 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. The distinct count is an important problem finding applications across a variey of disciplines like counting unique visitors to a website as well as common database operations (COUNT DISTINCT). 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 number of leading zeros (NLZ). The NTZ operation takes only 4 cycles on a dbcore 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 dbCores makes this dynamic scheduling essential to avoid long tail latencies. We also use the DMS to read and write buffers at peak bandwidth, and evaluate the performance o, talk about merge, hash functions used We can use a 64-bit hash function 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 (4-8 cycles), and we plan to address this in a future revision of our chip.

5.5 JSON Parsing

The JavaScript Object Notation (JSON) is an increasingly popular format amongst many applications that store and analyze large amounts of data. The JSON grammar is relatively simple, consisting of key-value pairs and a small number of syntactic tokens and datatypes (numbers, strings, lists, dictionaries), yet flexible because it supports nesting. Hence several applications use JSON to log data, as well as ingest JSON logs as an initial step in data analysis pipelines. We hence evaluate the efficiency of parsing JSON data using the DPU.

For representative baselines to compare to the DPU, we assume the source file is loaded into DDR memory (as with mmap() on x86). After evaluating open source C/C++ implementations of JSON (JSON11, RAPIDJSON, SAJSON) we selected SAJSON [3] as our best performing, portable baseline. Like several other parsers, it employs a switch-case structure and uses a single memory allocation to avoid

repeated overheads. For a benchmark, we populate JSON records with keys corresponding to the TPCH lineitems table. The datatypes hence consist of a mixture of integers, strings, dates and populate approximately 1GB of records. For this workload, SAJSON is able to achieve an IPC of 3.05, and parse the input data at 5.2 GB/s on our x86 machine. However, on the RAPID DPU, it only achieves a throughput of 645 MB/s âĂŤ even assuming all the workload records are perfectly prefetched into core-local SRAM(DMEM). The switch-case anatomy emits a large number of instructions, including several compare-branch with the default compiler (gcc, MIPS-O64). On the in-order RAPID dbCores operating at a relatively low frequency (1/3 compared to the x86, out-of-order machine), and lacking hardware branch prediction, the resulting processing cycles per byte (13.2) brings down the achievable memory bandwidth.

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. A table which supports looking up the input character in combination with the current parse state, and yield a pointer to a new parse action requires allocation of 256 (number of characters) $\times$ 8B (size of pointer to next parse action) $\times$ number of states in the grammar. Given JSONâĂŹs relatively small grammar (12 states), the parse table size fits within 23 KB. To allow concurrent processing on all dbcores, 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 dbcores, each dbcore allocates and reads an extra chunk. During parsing, the extra bytes are parsed as the last bytes of the dbcore processing the previous chunk and ignored by the dbcore which encounters them in its first chunk. Due to the efficiency of prefetching buffers using the DMS, this overhead is largely negligible.

Apart from using the DMS for moving data from DRAM to individual dbcore memories, further optimizations were required to comply with the requirements of parsing such as returning of pointers to full, null-terminated records. The straightforward approach of checking once for the end of a DMS descriptor, followed by a subsequent scan for the terminal character causes inputs to be read twice. Instead, we exploit the JSON grammar by adding an invalid byte at the end of the DMS buffer (like 0x00), force a parse error and invoke the appropriate next-state production rule with a single pass over the input. 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 dbcores, with an improvement of 8$\times$ in terms of performance/Watt over SAJSON.

5.6 Disparity

In this section, we evaluate a well-known computer vision workload that computes a disparity map [19]. Disparity map 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. In a nutshell, the disparity map workload 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. The resulting disparity map is essential component in many machine vision applications like autonomous cruising, pedestrian tracking, etc. This well-studied computer vision workload is known to be data intensive [22] whose memory accesses need to be carefully orchestrated to efficiently utilize the available memory bandwidth. In addition, the disparity computation involves four distinct image kernels each with different data access patterns that are not straight-forward to parallelize across multiple cores. These are the two main challenges that a developer faced in parallelizing this workload on any archiecture.

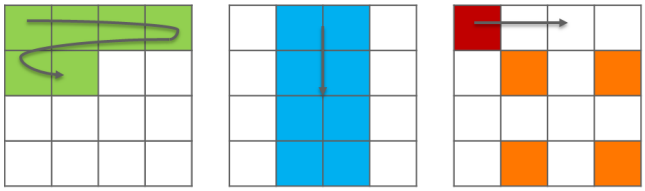


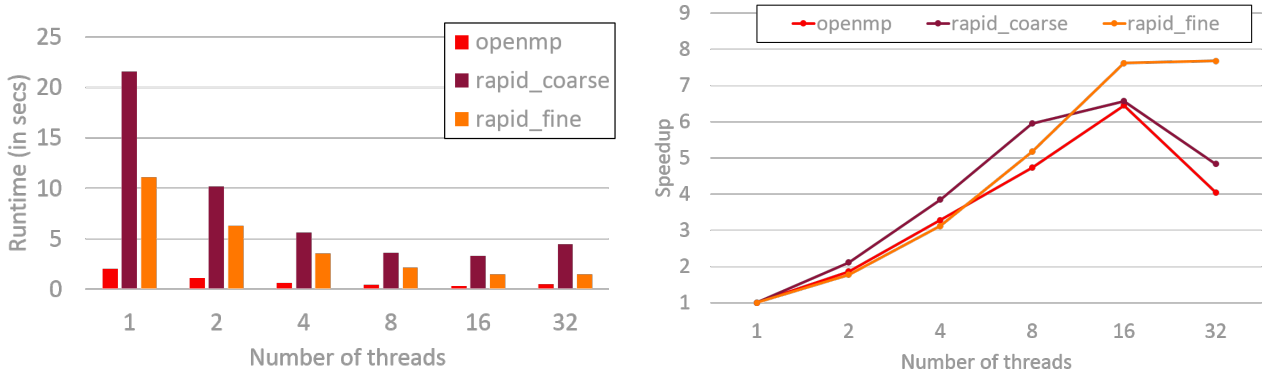
Figure 14: The three types of data access pattern in the disparity computer vision workload.

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 per each DbCore, where they cooperatively compute the disparity kernel in lockstep. Hence this approach might require 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 making each core independently compute disparity for a distinct shift in pixel of one of the images and finally aggregate the result across cores. Although this approach reduces the number of synchronization between the cores, may not efficiently utilize the available memory bandwidth. The result of this experiment is highly dependent on the architecture. For example, the cost of synchronization and ability to orchesterate memory accesses. Low-latency synchronization primitives and the DMS played a vital role in the RAPID architecture.

[ht]

The four disparity vision kernel involves three distinct data-access patterns as shown in the Figure-14. The most challenging data-access patterns are the columnar and pixelated-pattern. The software-managed DMEM via DMS makes orchestrating these access pattern significantly easier on the RAPID architecture. For instance, the most-challenging pixelated access pattern can be reduced into gathering pixels with two different strides into two sections of the DMEM scratch memory. Similarly, each core could independently fetch its own column into the core-private scratch memory.

We found that a tightly-coupled design of the RAPID architecture performed better under the fine-grained parallelism approach and resulted in a 2.9x slow down to 16 core IvyBridge Xeon machines (Oracle X4-2) with the OpenMP-based parallel implementation. This translates into 8.6x better perf/watt on RAPID compared to the X86 counterpart. The perfor-



(a) (b)

Figure 15: Performance of disparity workload on both X86 and RAPID, (a) shows the absolute runtime on each architecture, (b) shows scalability by plotting speedup over single thread performance with increasing number of threads. Both the graphs show fine- and coarse-grained parallelization approach on RAPID

mance result of scaling from 1 to 32 threads is shown in Figure-15. This shows that RAPID is a versatile system which benefits other non-SQL data-intensive workloads as well.

6. CONCLUSION

Rapid increases in data sizes requires a fundamental change in the way we tackle large scale analytics applications. We present the Database Processing Unit (DPU), a unique architecture focusing on memory bandwidth achieved/Watt, created to address the challenges of big data. We implement a variety of applications on the DPU, ranging from JSON Parsing to SQL operations, as well as a number of analytics workloads, and show an efficiency improvement of 5x to 15x in terms of performance/Watt. We note that this is a first step towards a larger rack scale system, where hundreds of thousands of DPUs are connected via a high bandwidth interconnect to provide an efficient solution to the exascale challenge.

7. REFERENCES

[1] S. R. Agrawal, C. M. Dee, and A. R. Lebeck, "Exploiting accelerators for efficient high dimensional similarity search," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 3:1–3:12. [Online]. Available: http://doi.acm.org/10.1145/2851141.2851144

[2] S. R. Agrawal, V. Pistol, J. Pang, J. Tran, D. Tarjan, and A. R. Lebeck, "Rhythm: Harnessing data parallel hardware for server workloads," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 19–34.

[3] C. Austin. (2013) Sajson: Single-allocation json parser. [Online]. Available: https://chadaustin.me/2013/01/single-allocation-json-parser

[4] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated network interfaces for high-bandwidth tcp/ip," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII. New York, NY, USA: ACM, 2006, pp. 315–324. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168897

[5] L. J. Cao, S. S. Keerthi, C.-J. Ong, J. Q. Zhang, U. Periyathamby, X. J. Fu, and H. P. Lee, "Parallel sequential minimal optimization for the training of support vector machines," *Trans. Neur. Netw.*, vol. 17, no. 4, pp. 1039–1049, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1109/TNN.2006.875989

[6] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011, software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[7] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[8] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 339–350. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325851.1325893

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA '11. New York, NY, USA: ACM, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[10] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *AofA: Analysis of Algorithms*, ser. DMTCS Proceedings, P. Jacquet, Ed., vol. AH. Juan les Pins, France: Discrete Mathematics and Theoretical Computer Science, Jun. 2007, pp. 137–156. [Online]. Available: https://hal.inria.fr/hal-00406166

[11] P. K. Gupta, "Accelerating datacenter workloads," 2016. [Online]. Available: http://www.fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf

[12] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 243–254. [Online]. Available: https://doi.org/10.1109/ISCA.2016.30

[13] T. H. Hetherington, M. O'Connor, and T. M. Aamodt, "Memcachedgpu: Scaling-up scale-out key-value stores," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 43–57. [Online]. Available: http://doi.acm.org/10.1145/2806777.2806836

[14] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt, "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, ser. ISPASS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 88–98.

[15] Intel, "Advancing moore's law in 2014 âĂŤ the road to 14 nm," 2014. [Online]. Available: http://www.intel.com/content/www/us/en/silicon-innovations/advancing-moores-law-in-2014-presentation.html

[16] N. Jouppi, "Google supercharges machine learning tasks with tpu custom chip," 2016. [Online]. Available: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html

[17] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[18] K.-J. Lin and J. D. Gannon, "Atomic remote procedure call," *IEEE Trans. Softw. Eng.*, vol. 11, no. 10, pp. 1126–1135, Oct. 1985. [Online]. Available: http://dx.doi.org/10.1109/TSE.1985.231860

[19] D. Marr and T. Poggio, "Cooperative computation of stereo disparity," in *From the Retina to the Neocortex*. Springer, 1976, pp. 239–243.

[20] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel efficient sparse matrix-matrix multiplication on multicore platforms," in *International Conference on High Performance Computing*. Springer, 2015, pp. 48–57.

[21] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678

[22] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 55–64.

[23] C. Vinschen and J. Johnston, "The red hat newlib c library." [Online]. Available: https://sourceware.org/newlib/

[24] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 249–260. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485944