A Data Layout Transformation (DLT) Accelerator:

Architectural support for data movement optimization in accelerated systems

Tung Thanh-Hoang[1], Amirali Shambayati[1], and Andrew A. Chien[1, 2]

[1]Department of Computer Science, University of Chicago, Chicago, Illinois, USA
[2]Argonne National Laboratory, Chicago, Illinois, USA
{hoangt, amirali, achien}@cs.uchicago.edu

***Abstract*—Technology scaling and growing use of accelerators makes optimization of data movement of increasing importance in all systems. Further, growing diversity in memory structures makes embedding such optimization in software non-portable. We propose an architectural solution, the Data Layout Transformation (DLT) that provides a simple set of instructions that enable software to describe the required data movement compactly, and free the implementation to optimize the movement based on knowledge of the memory hierarchy and system structure.**

The DLT architecture ideas can be applicable to both tradition and accelerator-based heterogeneous systems. Experiments show that DLT can make use of the full banwidth (>97%) of a wide range of memory systems, such as DDR3 and HMC, while incurring low software overhead (<2%) versus 11%-50% for advanced gather-scatter DMA engines.

We evaluate DLT in accelerated system, the 10x10 federated heterogeneous system, with DDR3 and HMC. Our results demonstrate that DLT improves system performance by 4.6x-99x (DDR3) and 4.4x-115x (HMC) as well as energy efficiency by 2.8x-48x (DDR3) and 1.4x-38x (HMC).

I. Introduction

The end of Dennard scaling [1] has risen *power wall* challenge [2] which is a primary motivation for the design of high performance energy efficiency systems in exascale computing era [3]. In general, system energy is composed of two major costs, one is consumed to process data and the other is spent to move/transform data in adequate destination/layout for computing units. On the one hand, the energy cost of data movement is significant, and even much more expensive than computation. As reported in [4], for 16nm process, the energy cost of data movement varies from 0.6x to 76x when moving data from SRAM/RF and DRAM respectively, compared to the energy cost of single Double-precision Fused Multiply Accumulator operation.

On the other hand, minimizing energy cost of data movement is essential for general-purpose architecture but it is eventually more critical for heterogeneous systems. The reason is, in heterogeneous systems, intensive computations can be easily accelerated by using multicore, GPUs or ASIC accelerators [5] etc. so the computation cost is extremely reduced. Therefore, the energy cost of data movement becomes a large fraction of system energy.

In reality, the memory hierarchy of main-stream processor is not designed for efficient data access. For example, accessing stride data or transforming data layout via caches suffer performance reduction and energy inefficiency due to low latency and data locality of caches. To overcome these bottlenecks of caches, scratchpad memory (or local memory) [6] has been used because it shows predictable performance, low latency and high energy efficiency. However, leveraging the advantage of local memory in system perspective would requires more investigations.

To date, there two challenging questions regarding the energy cost of data movement inside systems: *1)* How to minimize the amount of moving data? *2)* How to move data more efficiently?

There has been a number of prior works which tried to solve these questions, for example: *i)* latency avoidance techniques (e.g. SW/HW prefetcher [7, 8], multithreading architectures [9, 10]), *ii)* waste reduction techniques via redesigning memory systems (e.g. fine-grained access DRAM memory [11–14]) *iii)* hybrid techniques such as aggregaton of DMA engines [15], gather-scatter DMA engine [16], or adding logics for Processing-In-Memory (PIM) [17, 18].

Although these works have shown significant improvements to reduce the energy cost of data movement, they may incur following bottlenecks: data locality reduction due to missed prediction for prefetcher design efforts and implementation cost for memory system redesign, usage overhead due to the communication cost of co-processor DMA, bandwidth inefficiency due to interconnect contention when aggregate DMA engines, low programmability (e.g. PIM), or not fully utilize the local memory.

Therefore, we propose DLT architecture, a specialized accelerator toward to systematically minimize data movement and move data more energy efficiently. The key points of DLT architecture are simple hardware, instruction level integration, and memory-oriented design. As a result, the DLT accelerator has high programmability, high bandwidth efficiency and incurs low usage overhead which are critical to mitigate the energy limits of data movement inside accelerator-based heterogeneous systems. Evaluation shows good system performance and energy benefits when DLT is used to complement others accelerators in 10x10 architecture, a case study of federated heterogeneous system [19].

Specific contributions include:

1) Design of the Data Layout Transformation (DLT) instruction set architecture and implementation that can accelerate varied data movement (unit stride, non unit-stride, transpose operations) amongst vector register file, local memory, off-chip memory systems.
2) Evaluation of the DLT architecture that demonstrates its ability to support diverse memory systems and achieve their full bandwidth – 97% (DDR3) and 98% (HMC). For these systems at least, DLT's performance is only limited by memory interface and organization.
3) Evaluation of instruction execution overhead for data movement that shows DLT is efficient, achieving low overhead (<2%). This is better than gather/scatter DMA engines (8%-34%) widely used in commercial multicore DSPs. Further results show using DLT can increase overall performance by 11%-50% over that with the EDMA engine.
4) Finally, DLT really shines in highly-accelerated systems where memory-limited execution is common. Adding DLT to a 10x10 accelerated system [1] improves system performance by *4.6x-99x (DDR3), 4.4x-115x (HMC)* which yields *2.8x-48x (DDR3), 1.4x-38x (HMC)* energy efficiency.

[1]10x10 includes 64-sample/fixed 16-bit FFT, BnB (2Kbit SIMD vector machine) and MergeSort that can sort 1024 streams (1024 elements/stream)) [19, 20].

The this paper is organized as follows. Section II describes high-level system integration, microarchitecture and hardware implementation of the processor-integrated DLT accelerator. Section III describes the methodology, benchmark and explains configurations studied. The experimental results are presented in Section IV, followed by a discussion of the related work in Section V. We close with a summary and possible future directions in Section VI.

II. Microarchitecture, Programming Model and Hardware Implementation

A. Microarchitecture and High-level Integration

At the center of this work is the DLT accelerator whose microarchitecture and high-level integration are illustrated in Figure 1. In the high-level integration (Figure 1a), the DLT accelerator is tightly integrated with the pipeline of a simple RISC processor. Therefore, the DLT accelerator can take the advantage of instruction level integration such as high programmability, low usage overhead (reduce the cost to access system components).

In microarchitecture level (Figure 1b), the key component of DLT accelerator is DLT lanes. Each lane is composed of a buffer that keeps DLT instructions, address generation unit and a simple decoder. Each entry of buffer is 99 bits length which consists of instruction opcode (3b), source address (32b), destination address(32b) and DLT descriptor (32b) resulting in total 200B for buffer size. The address generation unit are (short) adders and subtracters which calculate new nextSrc, nextDst addresses and the decrement of nelem[i] (as the bookkeeper of individual gather/scatter instruction). These values are used to update buffer's fields under the control of ReadDone and WriteDone signals which are issued when an element is completely read (written) from (to) specific source (destination). The decoder generates enable read and write signals for memories and vector register file, and routes read/write addresses via multiplexers. When a DLT instruction executed by a DLT lane is completed (i.e. nelem[i]=0), that lane is released and can be assigned for a new instruction.

For parallelism, the DLT accelerator can simultaneously execute 4 gather/scatter instructions (i.e. at least 256/512 GB/s with DDR3/HMC request bandwidth) therefore it can saturate the peak bandwidth of memory systems while minimize hardware cost[2].

Listing 1: Using DLT intrinsic for 4k-FFT compute.

```
1 FFT32b_4k_1d (Opt, *src, *tdm4k, *dst) {
2 // Local memory allocation for src, dest, etc.
3 // Form necessary descriptors for DLT accelerator
4 desc = dlt_form_descriptor(64, 64, 8);
5 // Marshal phase : move data from DRAM to LocalMem;
6 dlt_gather(lm_src,  src,  desc);
7 dlt_gather(lm_tdm4k, tdm4k, desc);
8 dlt_gather_fence();
9 // Transpose phase
10 for (i=0; i<128; i+=2)
11   dlt_gather(lm_dst+i*64, lm_src+i, desc);
12 dlt_gather_fence();
13 // First run, using FFT accelerator for 64 rows
14   ... FFT computing
15 // Transpose phase
16  for (i=0; i<128; i+=2)
17    dlt_scatter(lm_dst+i, lm_src+i*64, desc);
18  dlt_scatter_fence();
19 // Second run, using FFT accelerator for 64 rows
20 }
```

[2]Buffers size is larger than the number of DLT lanes to reduce the cost of frequently calling fence instructions and provide compact code for long chain of gather/scatter instructions.

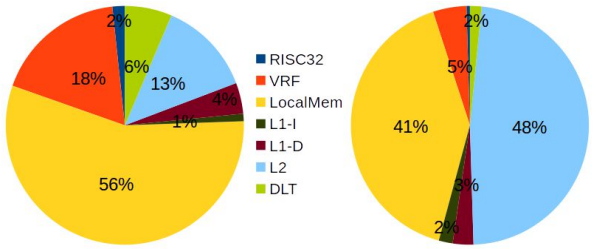
B. Programming Model

To make use of the DLT accelerator, we develop C-intrinsics that map directly to DLT instructions (see Table I). The *dlt_form_descriptor* intrinsic compacts the number of elements needed for gather/scatter, element stride and element size into a 32-bit descriptor. The *dlt_gather* and *dlt_scatter* intrinsics support gathering/scattering data between off-chip and local memory. Notice that, the address range of local memory is mapped into global address space thus *dlt_gather* and *dlt_scatter* intrinsics can be used to rearrange data (e.g. transposition) inside the local memory if both source and destination addresses locate in the local memory. Two other intrinsics *dlt_vgather* and *dlt_vscatter* can gather/scatter data between the local memory and vector registers.

We provide a set of memory synchronization (fencing) instructions in which *dlt_gather_fence*/*dlt_scatter_fence* can fence any memory instruction ahead until all concurrent *dlt_gather*/*dlt_scatter* instructions are completed to avoid memory inconsistency. Stronger *dlt_fence* instruction will fence all memory instructions ahead regardless issuing by either the RISC processor or the DLT accelerator. This instruction supports flushing data (dlt_flush) through cache hierarchy to off-chip memory before it is gathered/scattered. These features of instruction set of DLT accelerator can provide non-block programming model in addition to blocking model. Therefore the latency of data movement can be hidden under computation thus improving system performance.

For vector gather/scatter instructions (dlt_vgather/dlt_vscatter) are design in the same manner as memory gather/scatter instructions, except they are blocking since these types of data movement are usually fast. To demonstrate the usage of DLT accelerator, we show the code snippet of 4k 1D-FFT in Listing 1.

C. Hardware Implementation



(a) Power breakdown (total 1.25W) (b) Area breakdown (total 12.3 mm²)

Fig. 2: Hardware evaluation of processor-integrated DLT accelerator.

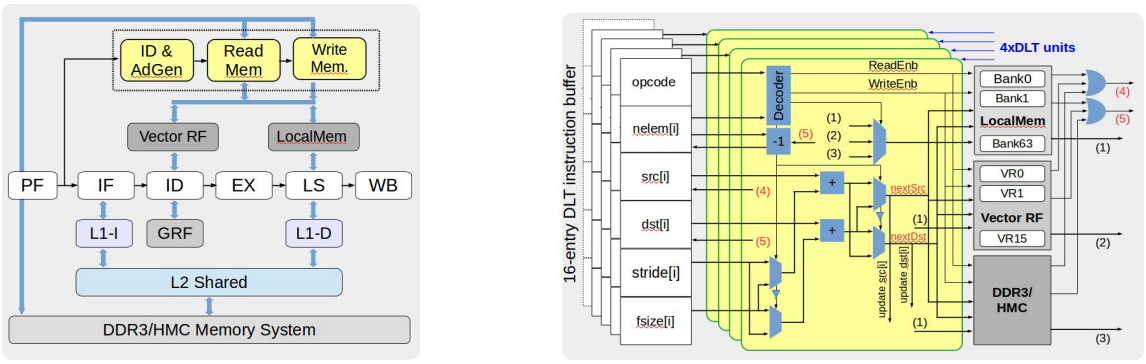
In order to evaluate the cost of integrating DLT, we design the DLT accelerator in high-level architecture description language (called LISA [21]). We use Synopsys Processor Designer [22] to compile the LISA code of DLT accelerator into Verilog which can be synthesized by using Synopsys Design Compiler tool with 32nm cell library. The area and power breakdown of DLT compared to other system components are shown in Figure 2. In the bottom line, the DLT accelerator takes only 6% (75mW) of system power and 2% (0.246 mm^2) of system area which are relatively low overheads.

III. Methodology and Experiments

A. System Simulation Platform

We build system simulation platform by integrating commercial and open-source tools. In the processor site, a core-processor, BnB and MergeSort accelerator[3] is designed by using LISA -*L*anguage for

[3]Except the FFT accelerator is generated by Spiral tool[23] for high clock rate then integrated into system simulator via LISA wrapper.



(a) High-level integration of DLT accelerator.

(b) Microarchitecture.

Fig. 1: The high-level integration (Figure 1a) and microarchitecture of DLT accelerator (Figure 1b, *(4)* is *ReadDone* signal that enables updating *src[i]* by *nextSrc*, *(5)* is *WriteDone* signal that enables updating *dst[i]* by *nextDst* and decreasing *nelem[i]*).

TABLE I: Micro-Instruction Set Architecture of the DLT accelerator.

Intrinsic	Instruction	Functional description
int dlt_form_descriptor(int nelem, int stride, int fsize)	FORMDESC R1 R2 R3	Pack (number_of_elems, stride, elem_size) into a 32-bit descriptor
void dlt_gather(ptr* dstA, ptr* srcA, int desc)	GATHER R1 R2 R3	Gather data described by *desc* from *srcA* to *dstA*
void dlt_scatter(ptr* dstA, ptr* srcA,int desc)	SCATTER R1 R2 R3	Scatter data described by *desc* from *dstA* to *dstA*
void dlt_vgather(vrDst, ptr* srcA,int desc)	VGATHER V1 R1 R2	Gather data described by *desc* from *srcA* to (*vrDst*
void dlt_vscatter(ptr* dstA, vec vrSrc,int desc)	VSCATTER R1 V1 R2	Scatter data described by *desc* from *vrSrc* to *dstA*
void dlt_gather_fence()	GATHERFENCE	Avoid execution of memory instructions until all concurrent *dlt_gather* instructions complete
void dlt_scatter_fence()	SCATTERFENCE	Avoid execution of memory instructions until all concurrent *dlt_scatter* instructions complete
void dlt_flush(ptr* addr1,ptr* addr2)	FLUSH R1 R2	Flush and invalidate all the cache lines within address range defined by (*addr1, addr2*)
void dlt_fence()	FENCE	Avoid execution of ALL memory instructions.

(*) The full ISAs of FFT, MergeSort, and BnB accelerators are described in [19].

*I*nstruction *S*et *A*rchitcture- which can be simulated in cycle-accurate level or compiled into synthesizable Verilog using Synopsys Processor Design [22]. The cycle-accurate simulation provides the instruction and cycle counts that then calibrate using the power of core and accelerators, extracted from synthesis with 32nm standard cell library.

Regarding cache hierarchy and local memory modelings, we calibrate using power generated by using Cacti [24]. Then we integrate a functional and timing model for cache hierarchy (extracted from the MarssX86 simulator [25]) with the Synopsys tools. This gives us a reasonable simulation runtime and more accurate evaluation.

The last component of our simulation platform is off-chip memory system, we use cycle-accurate DRAMSim2 tool [26] to evaluate the traditional DDR3 and comprehensive extension to model modern Hybrid Memory Cube [13] which provides package-based interface for 3D-stacked DRAM-based memory [27]. The HMC has been known as one of the most high throughput and energy efficient memory system. We have validated the performance and energy models of HMC against [28]. The configuration of the system simulation is shown in Table II.

B. Evaluated Designs and Applications

We consider the advantage of DLT accelerator in combination of one of other three FFT, BnB and Sort accelerators. For each benchmark evaluation, we study the following configurations textita) **Baseline** is designed without any accelerator support *b)* **Accel** has one of FFT, BnB or Sort accelerator and but no DLT added *c)* **DLT+Accel** consists of DLT and one of other accelerators.

For benchmark selection, we use totally 5 fundamental applications which are widely-used in domain-embedded systems: *i)* 2D Fast Fourier Transformation (**2D-FFT**) accelerated by FFT accelerator

TABLE II: Simulation Platform Configuration.

Parameter	Value
Processor	32nm, TSMC-based
Chip clock rate	1Ghz
Core type	In-order, MIPS-like ISA, 5-stage pipeline
Vector register file	256B x 16 registers
Local memory	64 banks and 256B IO (4Bx16k entry/bank)
Cache hierarchy	L1-I: 32KB, 2-cycle latency
	L1-D: 24KB, 2-cycle latency
	Shared L2: 512KB, 10-cycle latency
Main memory	
DDR3	2GB, 4-rank, 8-device, 667 Mhz, 10.6 GB/s
HMC	4GB, 4-rank, 8-device, 1.25 Ghz, 40 GB/s, 3D-stack

ii) Discrete Wavelet Transform (**DWT**), 2D-Convolution (**2D-Conv**) and Matrix Multiplication (**MatMult**) accelerated by BnB accelerator *iii)* Merge-sort (**MergeSort**) supported by Sort accelerator. Moreover, each configuration will be evaluated with DDR3 and HMC memory. Table III summarizes the system configurations and applications mapped to evaluated accelerators.

IV. Experimental Results

In this section, we demonstrates *i)* the bandwidth efficiency of DLT accelerator across DDR3 and HMC memory *ii)* the usage overhead of DLT accelerator and state-of-the-art gather-scatter DMA engine used in multicore DSP chip *iii)* the performance and energy benefits of using DLT accelerator to augment 10x10 accelerator-based heterogeneous system.

A. Bandwidth Efficiency of the DLT Accelerator

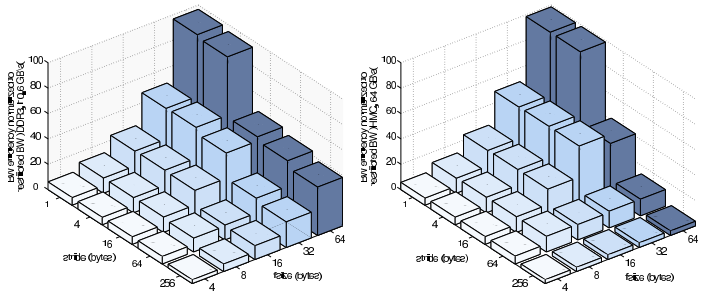
First, we study the memory bandwidth efficiency achieved by the DLT accelerator with DDR3 and HMC memory. By varying *stride* we

TABLE III: Summary of evaluated design and benchmark (Here, LM=Local Memory, MM=Main Memory and VR=Vector Register).

Application (*)	Workload size	Type of data movement	Design	
			Accel	DLT+Accel
2D-FFT	4kx4k, fixed 16-bit sample	Gather/scatter data between LM and MM, and transpose data inside LM	FFT	DLT+FFT
DWT	1080x1920, int	Gather/scatter data between LM and MM, and transpose data inside LM	BnB	DLT+BnB(**)
2D-Conv	1080x1920, int	Gather/scatter data between LM and MM	BnB	DLT+BnB(**)
MergeSort	1k streams, 1k int/stream	Gather/Scatter data between LM and MM	Sort	DLT+Sort
MathMult	4kx4k, int	Gather/Scatter data between LM and MM, LM and VR	BnB	DLT+BnB

(*) All applications are implemented in tile-based fashion thus data can fit in local memory for fast streaming to accelerators.
(**) Latency of data movement is hidden under computation.

can evaluate the impacts of memory array organization. Meanwhile, changing *fsize*, as the payload of DLT request, can expose the limit of memory interface. Before discussing the bandwidth efficiency of DLT, we define *restricted bandwidth* as maximum bandwidth that can be achieved for interface between the DLT and memory system.



(a) DDR3. (b) HMC.

Fig. 3: Bandwidth efficiency of DLT varying stride and element size.

Beginning with DDR3 memory, Figure 3a shows the bandwidth efficiency of DLT accelerator normalized to restricted bandwidth, in this case, that is 10.6 GB/s as the peak bandwidth of DDR3. We observe the bandwidth efficiency of DLT increasing for small *stride* and large *fsize* values. The maximum bandwidth efficiency is 97% (i.e. 10.3 GB/s when *stride=1* and *fsize=64*) which nearly saturates DDR3 peak bandwidth. The reason is the DLT accelerator can utilize the interface of DDR3 memory (64B payload/request) and incur low impacts of memory array organization due to unit-stride data access. On the other hand, the bandwidth efficiency of DLT is decreased for large *stride* and/or small *fsize* values. Given page size defined by memory array organization, large *stride* value causes high frequent opening and closing memory pages resulting in the increasing of access time. Meanwhile, small *fsize* value implies that memory interface is not fully utilized or less useful data can be gathered per request.

Moving to the HMC memory, Figure 3b shows the bandwidth efficiency of the DLT accelerator normalized to the restricted bandwidth which is 64 GB/s since HMC can request maximum 64B payload with 1GHz clock. The bandwidth efficiency of DLT is increased up to 98% (63.07 GB/s) of the restricted bandwidth as the best case (i.e. *stride=1* and *fsize=64*). However, it should be noted that the HMC memory has peak bandwidth up to 128 GB/s [13]. Therefore, to utilize the peak bandwidth of HMC, the DLT accelerator should send requests with 128B payload (also supported by DLT) for unit-stride access[4].

In summary, the DLT accelerator archive high full bandwidth - 97% (DDR3), 98% (HMC)- by selecting proper descriptors to utilize memory interface and mitigate the bottlenecks of memory array.

[4]We use DLT request with 64B payload to consider if limited memory interface can affect the bandwidth efficiency of DLT accelerator.

B. Usage Overhead Comparison of the DLT and EDMA Engine

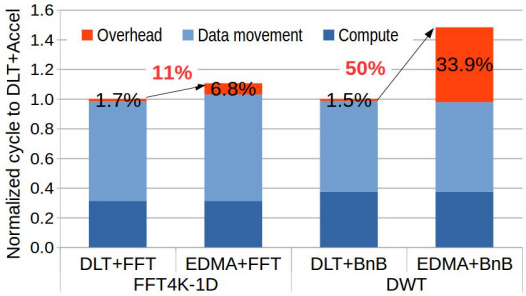


Fig. 4: Cycle count comparison between the DLT and EDMA engine.

In this section, we show that the DLT accelerator has lower usage overhead (in terms of cycle count) than the gather-scatter DMA (EDMA [16]) engine that increasing overall system performance.

The EDMA engine is used in the latest multicore TI-DSP that support more types of data movement than the DLT accelerator (e.g. data movement from/to multi-level cache). However, the EDMA engine is complicated hardware (in both logic and interconnection) thus it exists, in system, as co-processor and communicate with CPU via interrupts that increase usage overhead (e.g. for setup and check completion of EDMA gather/scatter).

We estimate the total cycle count of system as the summation of computation and data movement costs. For computation cost, we assume that both systems using EDMA engine and DLT accelerator use accelerators (e.g. FFT, BnB, and MergeSort) thus they have the same cycle count for computation which is reported by simulator.

Since the detail architecture of EDMA engine and its interconnection are not embodied thus we estimate the data movement cost of system using EDMA as the summation of cycle overhead due to using EDMA and cycle count needed for moving data. The former is estimated as the sum of cycle overhead of all data movement instructions using EDMA. [5]. The latter is the sum-of-products of maximal bandwidth [6] achieved by EDMA for particular type of data movement multiplied by corresponding amount of moving data that is easy estimated from workload size. For data movement cost of using DLT, we obtain this number from system simulator.

Figures 4 shows the cycle count of using DLT accelerator and EDMA engine (normalized to total cycle of system using DLT) for 4k-sample/16-bit 1D-FFT and 1080x1920/32-bit DWT applications with DDR3 memory.

For the 4k-FFT application, the EDMA engine incurs 6.8% usage overhead which is four times larger than using the DLT accelerator

[5]Cycle overhead of EDMA for transposition and gather/scatter are 291 and 301 cycles respectively [29].

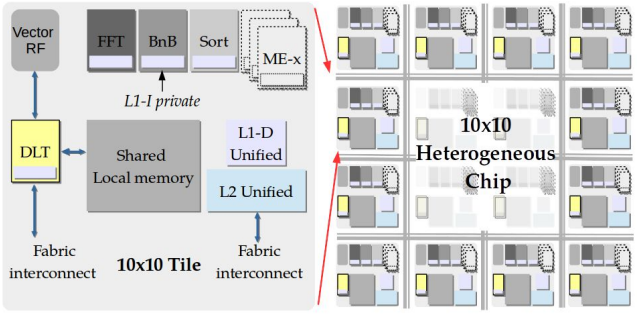
[6]Maximum bandwidth of EDMA for transposing and gather/scatter are 7.8 GB/s and 5.3 GB/s respectively [29].

(1.7%). This overhead increases the cycle count of system using the EDMA engine by total 11%[7] compared to one using the DLT accelerator. The reason here is DLT instructions to setup descriptor (i.e. *dlt_form_descriptor*) and check completion (i.e. *dlt_gather_fence* and *dlt_scatter_fence*) take only few cycles (∼4) which is the advantage of instruction level integration. On the contrast, the EDMA engine requires much more cycles to setup long descriptors (32B) and check completion after moving data (i.e. backup and restore registers for handling interrupt).

Moving to the DWT application, the usage overhead of DLT accelerator is only 1.5% which is much less than using the EDMA engine (33.9%). As a result, system using the DLT accelerator shows 50% cycle count reduction compared to one using the EDMA engine. The main reason here is DWT benchmark requires many requests which just move small amount of data, therefore EDMA engine is more frequently used resulting in the increasing of cycle overhead, as a large fraction of total cycle count).

It should be noted that the bandwidths of EDMA engine used in our evaluation are optimal values thus its estimation for cycle count of data movement and cycle usage overhead are eventually overestimated. Meanwhile, DLT accelerator is evaluated by using cycle-accurate simulator. Therefore, our cycle count comparison is *confident and fair* which demonstrates the low overhead (and performance benefits) of using the DLT accelerator over the EDMA engine.

C. Performance and Energy Benefits of DLT for 10x10 Architecture



(a) DLT integration in 10x10 core. (b) 10x10 heterogeneous arch.

Fig. 5: Integration of DLT in 10x10 heterogeneous architecture.

The energy inefficiency of data movement becomes critical for accelerator-based heterogeneous systems where memory-limited execution is common [4]. In this section, we demonstrate that the DLT accelerator has high performance and energy benefits when using as a complement of other accelerators in the 10x10 architecture [19].

Beginning with the DDR3 memory, Figures 6a and 6c show that the performance and energy efficiency of 10x10 architecture using the DLT accelerator (**DLT+Accel**) are respectively improved by 4.6x-99x and 2.8x-48x compared to system without DLT support (**Accel**). Among evaluated applications, MatrixMult has lowest improvements of performance and energy efficiency than the others because it is highly compute-intensive that can reduce the benefits of DLT (with both DDR3 and HMC memory). On the contrast, FFT and MergeSort are data-intensive applications thus using DLT shows highest performance and energy efficiency than the others.

Moving to the HMC memory, the system performance and energy efficiency of **DLT+Accel** design are respectively increased as much as 4.4x-115x (Figure 6b) and 1.4x-39x (Figure 6d) compared to the **Accel** design which is not supported by DLT. Notice that, the energy benefit of **DLT+Accel** design with HMC memory is less than one with DDR3. This is because HMC memory is more energy efficient than DDR3 thus the energy cost of data movement with HMC, as a fraction of system energy, is smaller.

For geometry mean across all applications, the performance and energy efficiency of **DLT+Accel** design are gained by 17x (DDR3), 19x (HMC) and 7.1x (DDR3), 5.1x (HMC) compared to the **Accel**.

Finally, when directly comparing the **DLT+Accel** and **Baseline** system, we observed further benefits of using DLT. More detail, using DLT in the **DLT+Accel** design can gain system performance by 46x-1464x (DDR3), 44x-1876x (HMC) and energy efficiency by 14x-329x (DDR3), 10x-374x (HMC) compared to the **Baseline**.

V. Related Work

In this section, we consider DLT accelerator in the context of related works which are divided into the following categories

Latency avoidance techniques avoid movement latency by either moving data in advance or rapidly transforming data layout for efficient access. In the former group, the most well known technique is data prefetch which predicts future memory addresses then fetches adjacent or stride cache lines into caches/buffers by using software [30] or hardware prefetchers [31]. Another technique is decoupling memory access and execution by using a dedicated processor to fetch data ahead [32]. While the latter group leverages multithreading architectures (e.g. Xeon Phi [9], GPU [10]) to utilize memory bandwidth for data transposition thus improving overall system performance.

Although aforementioned techniques are feasible and efficient, they may have some bottlenecks. For example, miss rate of predictor in prefetchers, long latency and low bandwidth utilization of caches in multithreading architectures[8] can cause performance loss. Meanwhile, the energy cost of hardware predictors, software-based address generation (Xeon Phi, GPU) can affect system energy efficiency.

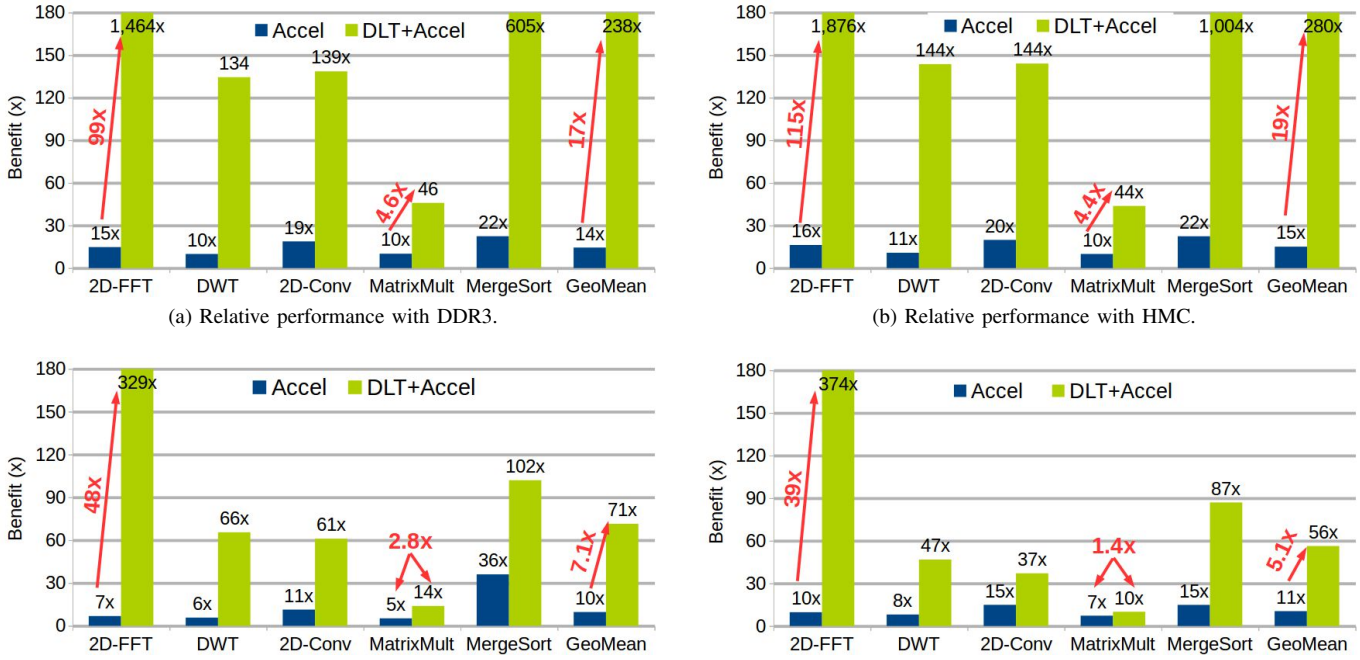
On the contrast, our DLT architecture is relatively simple and efficient because it exploits the facts that most access patterns are regular (e.g. in domain-specific applications) and easily represented by DLT's descriptor in programming level (therefore, no need complex predictors). Furthermore, our DLT focuses on data movement involving the local memory which has lower latency and access energy, higher bandwidth than caches therefore it can simultaneously improve system performance and energy efficiency.

Waste reduction techniques eliminate moving unnecessary data between system components in order to improve performance and energy efficiency. These techniques focus on redesigning either the array organization or interface of main-stream memory systems which is not optimized to access stride data [33]. For redesign memory, adaptive memory systems has been proposed to support fine-grained accesses by exploiting granularity information (e.g. cache structure, sub-rank array) provided by software (e.g. OS) [11] or hardware [12]. Memory interface can be redesigned to support packet-based protocol which allows accessing data with variable size. For example, payloads in variable size can be encapsulated (BOB [14], HMC [13]) into packets using the advanced logic die of memory systems. Yet our DLT architecture is designed to supports packet-based interface for waste data reduction.

Hybrid techniques combine latency avoidance and waste reduction mechanisms. The conventional DMA engines have been widely

[7]Extra 4.2% cycle overhead is due to the bandwidth of data movement between the local memory and DDR3 using EDMA is only one half (5.3 GB/s) of DDR3 peak bandwidth (10.6 GB/s).[16].

[8]Bandwidth utilization of Xeon Phi is 2.5% for matrix transposition (3.2x less than using DLT).



(a) Relative performance with DDR3.

(b) Relative performance with HMC.

(c) Energy benefit with DDR3.

(d) Energy benefit with HMC.

Fig. 6: Performance and energy benefits (normalized to **Baseline** design) of DLT accelerator for the 10x10 architecture.

used in SoC design to move data from/to off-chip memory. For example, Cell processor[15] is an advanced architecture which can aggregate the bandwidth of multiple DMA engines to accelerate data movement. To us, Cell architecture shares some ideas with DLT but it have some bottlenecks compared to the DLT accelerator such as: not support stride access in single DMA instruction, the aggregation of DMA engines is limited by interconnection contention.

Furthermore, we consider powerful gather-scatter EDMA engine [29] on contrast to the DLT accelerator. Although EDMA engine can cover a wide range of data movements (e.g. gather/scatter for caches) by flexible descriptors (e.g. linking/chaining), it is really complicated hardware and existed as co-processor. Hence, the usage overhead of EDMA engine may affect system performance, for example, in applications which requires many requests to move small data size.

Finally, some works add specialized hardware into logic die of memory to support layout transformation near memory [17, 18]. In overall, these approaches require design efforts and target only in-memory data movements which are unlike the DLT accelerator.

VI. Summary and Future Work

This paper presented Data-Layout-Transform (DLT), a memory-oriented, high programmable accelerator to optimize data movement across system components. Our result demonstrates that the proposed DLT accelerator achieves high bandwidth efficiency, as 97% (DDR3) and 98% (HMC), for varied memory systems and low usage overhead (<2%) compared to the advanced gather-scatter DMA engine. Moreover, the DLT accelerator can complement the accelerators of 10x10 heterogeneous architecture to increase system performance and energy efficiency as much as 4.6x-99x (DDR3), 4.4x-115x (HMC) and 2.8x-48x (DDR3), 1.4x-38x (HMC) respectively.

Interesting future works include: compiler support to detect the code spots to apply DLT intrinsics which is helpful to simplify programmers job, support 2D transpose with single instruction, express complex access patterns in single packet of memory system, or emsemble multiple DLT accelerators for rapid data movement.

VII. Acknowledgment

Funding of this work was provided in part by the Defense Advanced Research Projects Agency under award HR0011-13-2-0014 and the NSF under award NSF OCI-10-57921. We acknowledge Dilip P. Vasudevan and Yuanwei (Kevin) Fang at UChicago for their contributions to BnB and Sort micro-engines respectively. Finally, we thank Synopsis for generous university program support.

References

[1] R Dennard. “Design of Ion-implanted MOSFETs with Very Small Physical Simensions”. In: *JSSC* (1974).

[2] H. Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *ISCA*. 2011.

[3] J. Shalf et al. “Exascale Computing Technology Challenges”. In: *VECPAR*. 2011.

[4] B. Dally. *NVIDIA’s Path to ExaScale*. GPU Technology, SC. 2014.

[5] C. Cascaval et al. “A taxonomy of accelerator architectures, their programming models”. In: *J. of IBM Research and Development* (2010).

[6] S. Kayla et al. “Efficient HPC Data Motion via Scratchpad Memory”. In: *DISCS*. 2012.

[7] R. M. Rabbah et al. “Compiler Orchestrated Prefetching via Speculation and Predication”. In: *ASPLOS*. 2004.

[8] Y. Ishii et al. “Access Map Pattern Matching for Data Cache Prefetch”. In: *ICS*. 2009.

[9] V. Andrey. *Multithreaded Transposition of Square Matrices with Common Code for Intel Xeon Processors and Intel Xeon Phi Co-Processors*. Download link http://research.colfaxinternational.com. 2013.

[10] I.-J. Sung et al. “In-place Transposition of Rectangular Matrices on Accelerators”. In: *PPoPP*. 2014.

[11] D. H. Yoon et al. “Adaptive Granularity Memory Systems: A Tradeoff Between Storage Efficiency and Throughput”. In: *ISCA*. 2011.

[12] D. H. Yoon et al. “The Dynamic Granularity Memory System”. In: *ISCA*. 2012.

[13] HMC-Consortium. *Hybrid Memory Cube Spec. v1.1*. Tech. rep. 2014.

[14] E. Cooper-Balis et al. “Buffer-on-Board Memory Systems”. In: ISCA. 2012.

[15] S. Williams et al. “Scientific Computing Kernels on the Cell Processor”. In: *Int. J. Parallel Program.* (2007).

[16] N. Gilbert. “Exploiting DMA for Performance and Energy Optimized STREAM on a DSP”. In: *HPPAC Workshop, joined with IPDPS*. 2014.

[17] L. Zhang et al. “The Impulse Memory Controller”. In: *IEEE Trans. Comput.* (2001).

[18] Q. Guo et al. “3D-Stacked Memory-Side Acceleration: Accelerator and System Design”. In: *WoNDP, joined with MICRO*. 2014.

[19] A. C. Andrew et al. “10x10: A Case Study in Federated Heterogeneous Architecture”. In: *Submitted*. 2015.

[20] H. T. Tung et al. “Performance and Energy Limits of a Processor-integrated FFT Accelerator”. In: *HPEC*. 2014.

[21] V. Zivojnovic et al. “LISA-machine Description Language and Generic Machine Model for HW/SW Co-design”. In: *VLSI Signal Processing*. 1996.

[22] Synopsys. *Processor Designer*. Online, http://www.synopsys.com.

[23] P. Milder et al. “Computer Generation of Hardware for Linear Digital Signal Processing Transforms”. In: *ACM TODAES* (2012).

[24] M Naveen et al. *CACTI 6.0: A Tool to Model Large Caches*. Tech. rep. HP Lab., 2009.

[25] A. Patel et al. “MARSS: A Full System Simulator for Multicore x86 CPUs”. In: *DAC*. 2011.

[26] P. Rosenfeld et al. “DRAMSim2: A Cycle Accurate Memory System Simulator”. In: *Computer Architecture Letters* (2011).

[27] C. Ke et al. “CACTI-3DD: Architecture-level modeling for 3D die-stacked DRAM main memory”. In: *DATE*. 2012.

[28] B. Shekhar et al. “Exascale Computing - A Fact or a Fiction ?” In: *Technical talk at IPDPS*. 2013.

[29] TI. *TMS320C6678 Memory Access Performance*. Tech. rep. 2011.

[30] T. C. Mowry et al. “Design and Evaluation of a Compiler Algorithm for Prefetching”. In: *ASPLOS*. 1992.

[31] V. Jiménez et al. “Making Data Prefetch Smarter: Adaptive Prefetching on POWER7”. In: *PACT*. 2012.

[32] J. E. Smith. “Decoupled Access/Execute Computer Architectures”. In: *ACM Trans. Comput. Syst.* (1984).

[33] R. C. Murphy et al. “On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications”. In: *IEEE Trans. Computer* (2007).