Bachelors Thesis

Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Anatol Constantin Fürst

16th January 2024

Technische Universität Dresden
Faculty of Computer Science
Institute of Systems Architecture
Chair of Operating Systems

Academic Supervisors:
Prof. Dr.-Ing. Horst Schirmeier
Prof. Dr.-Ing. habil. Dirk Habich
M.Sc. André Berthold

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Systemarchitektur, Professur für Betriebssysteme

Aufgabenstellung für die Anfertigung einer Bachelor-Arbeit

Studiengang:	Bachelor
Studienrichtung:	Informatik (2009)
Name:	Constantin Fürst
Matrikelnummer:	4929314
Titel:	Data Movement in Heterogeneous Memories with Intel Data Streaming Accelerator

Developments in main memory technologies like Non-Volatile RAM (NVRAM), High Bandwidth Memory (HBM), NUMA, or Remote Memory, lead to heterogeneous memory systems that, instead of providing one monolithic main memory, deploy multiple memory devices with different non-functional memory properties. To reach optimal performance on such systems, it becomes increasingly important to move data, ahead of time, to the memory device with non-functional properties tailored for the intended workload, making data movement operations increasingly important for data intensive applications. Unfortunately, while copying, the CPU is mostly busy with waiting for the main memory, and cannot work on other computations. To tackle this problem Intel implements the Intel Data Streaming Accelerator (Intel DSA), an engine to explicitly offload data movement operations from the CPU, in their newly released Intel Xeon CPU Max processors.

The goal of this bachelor thesis is to analyze and characterize the architecture of the Intel DSA and the vendor-provided APIs. The student should benchmark the performance of Intel DSA and compare it to the CPU's performance, concentrating on data transfers between DDR5-DRAM and HBM and between different NUMA nodes. Additionally, the student should find out in what way and to what extent parallel processes copying data interfere with each other. Analyzing the performance information, the thesis should outline a gainful utilization of the Intel DSA and demonstrate its potential by extending the Query-driven Prefetching concept, which aims to speed up database query execution in heterogeneous memory systems.

Gutachter:	Prof. Dr.-Ing. Dirk Habich
Betreuer:	André Berthold, M.Sc.
Ausgehändigt am:	4. Dezember 2023
Einzureichen am:	19. Februar 2024

Prof. Dr.-Ing. Horst Schirmeier
Betreuender Hochschullehrer

Statement of Authorship

I hereby declare that I am the sole author of this master thesis and that I have not used any sources other than those listed in the bibliography and identified as references. I further declare that I have not submitted this thesis at any other institution in order to obtain a degree.

Dresden, 16th January 2024

Anatol Constantin Fürst

Abstract

...abstract ...

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction to Querry driven Prefetching

- database context where we have an execution plan for a querry to be executed
- use knowledge about the querries sub-tasks to determine part of the table which is worth to cache (used multiple times)
- refer to whitepaper for more information

1.2 Introduction to Intel Data Streaming Accelerator

-

1.3 Goal Definition

- use DSA to offload asynchronous prefetching tasks
- prefetch into HBM which is smaller (cant hold all data) but faster (used as large cache)
- effect is lower cpu utilization for copy
- this allows to focus on actual pipeline execution

2 Technical Background

> Intel DSA is a high-performance data copy and transformation accelerator that will be integrated in future Intel® processors, targeted for optimizing streaming data movement and transformation operations common with applications for high-performance storage, networking, persistent memory, and various data processing applications. [1, p. 15]

Introduced with the 4th generation of Intel Xeon Scalable Processors [2], the DSA promises to alleviate the CPU from 'common storage functions and operations such as data integrity checks and deduplication' [2]. This chapter will give an overview of the architecture, software and the interaction of these two components. The reader will be familiarized with the setup and equipped with the knowledge to configure the system for a specific use case.

consider adding projected use cases as in the architecture specification here

2.1 Architecture

To be able to optimally utilize the Hardware, knowledge of its workings is required to make educated decisions. Therefore, this section describes both the workings of the DSA engine itself and the view that is presented through software interfaces. All statements are based on Chapter 3 of the Architecture Specification by Intel [1].

2.1.1 Hardware Architecture

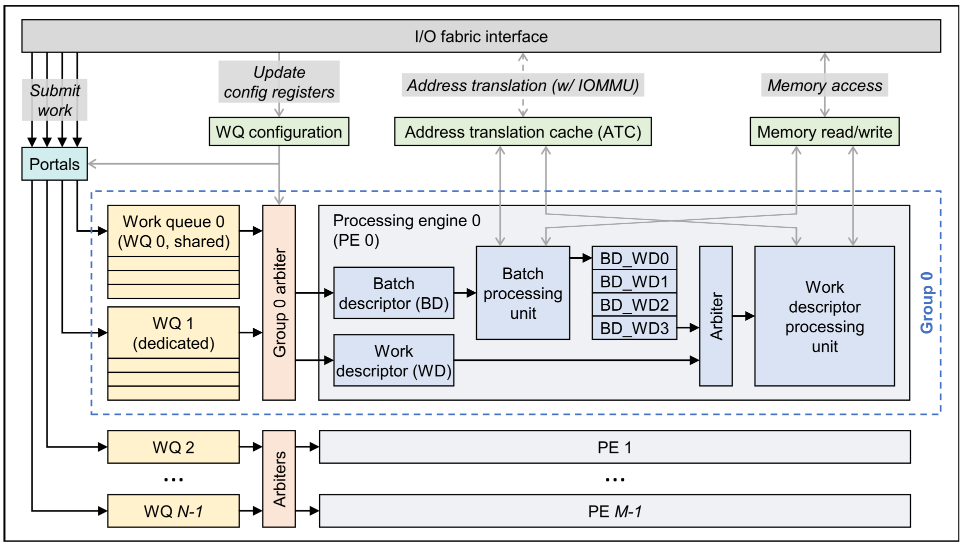

Figure 2.1:
Internal Archtiecture Block Diagramm
Taken from Figure 1a of [3]

The accelerator is directly integrated into the Processor and attaches via the I/O fabric interface over which all communication is conducted. Over this interface, it is accessible as a PCIe device. Configuration therefore is done through memory-mapped registers set in the devices Base Address Register (BAR). Through these, the devices layout is defined and memory pages for work submission are set. In a system with multiple processing nodes, there may also be one DSA per node.

To satisfy different use cases, as already mentioned, the layout of the DSA may be software-defined. The structure is made up of three components, namely Work Queue (WQ)s, Engines and Groups. WQs provide the means to submit tasks to the device and will be described in more detail shortly. An Engine is the processing-block that connects to memory and performs the described task. Using Groups, Engines and WQs are tied together. This means, that tasks from one WQ may be processed from multiple Engines and that vice-versa, depending on the configuration. This flexibility is achieved through the Group Arbiter which connects the two components and acts according to the setup.

A WQ is accessible through so-called portals, which are mapped memory regions. Submission of work is done by writing a descriptor to one of these portals. A descriptor is 64 Byte in size and may contain one specific task (task descriptor) or the location of a task array in memory (batch descriptor). Through these portals, the submitted descriptor reaches a queue of which there are two types with different submission methods and use cases. The Shared Work Queue (SWQ) is intended to provide synchronized access to multiple processes and each group may only have one attached. A PCIe Deferrable Memory Write Request (DMR), which guarantees implicit synchronization, is generated via x86 Instruction ENQCMD and communicates with the device before writing. This results in higher submission cost, compared to the Dedicated Work Queue (DWQ) to which a descriptor is submitted via x86 Instruction MOVDIR64B. The DWQ is therefore more performant but may require access control mechanisms and may only be accessed by one process at a time.

To handle the different descriptors, each Engine has two internal execution paths. One for a task and the other for a batch descriptor. Processing a task descriptor is straightforward, as all information required to complete the operation are contained within. For a batch, the DSA first reads the batch descriptor, then fetches all task descriptors for the batch from memory and processes them. An Engine can also trigger a page fault when trying to access an unloaded page and wait on its completion, if configured to do so. Otherwise, an error will be generated in this scenario.

Ordering of operations is only guaranteed for a configuration with one WQ and one Engine in a Group when submitting exclusively batch or task descriptors but no mixture. Even then, only write-ordering is guaranteed, meaning that ‘reads by a subsequent descriptor can pass writes from a previous descriptor’ [1, p. 30]. A different issue arises, should an operation fail: the DSA will continue to process the following descriptors. Care must therefore be taken with read-after-write scenarios, either by waiting for a successfull completion before submitting the dependant, inserting a drain descriptor for tasks or setting the fence flag for a batch. The latter two methods tell the processing engine that all writes must be commited and, in case of the fence in a batch, abort on previous error.

An important aspect of modern computer systems is the separation of address spaces through virtual memory. The DSA must therefore handle address translation, as a process submitting a task will not know the physical location in memory which causes the descriptor to contain virtual values. For this, the Engine communicates with the Input/Output Memory Management Unit (IOMMU) and Address Translation Cache (ATC) to perform this operation. For this, knowledge about the submitting processes is required, and therefore each task descriptor has a field for the Process Address Space ID (PASID) which is filled by the ENQCMD instruction for a SWQ or set statically after a process is attached to a DWQ.

The completion of a descriptor may be signaled through a completion record and interrupt, if configured so. For this, the DSA ‘provides two types of interrupt message storage: (1) an MSI-X table, enumerated through the MSI-X capability; and (2) a device-specific Interrupt Message Storage (IMS) table’ [1, p. 27].

2.1.2 Software View

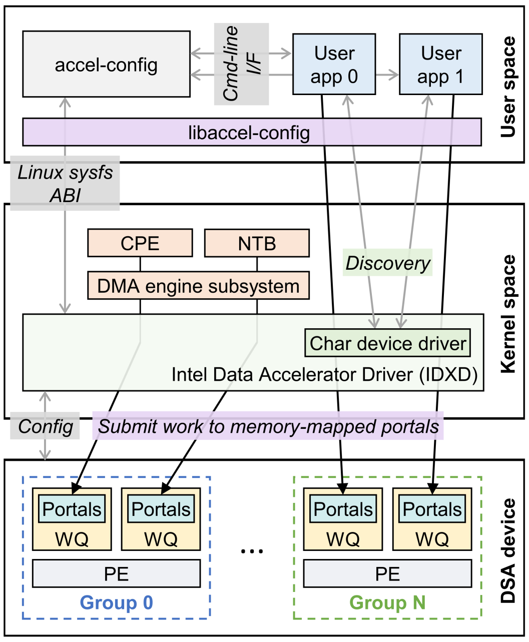

Figure 2.2:
Software View Block Diagramm
Taken from Figure 1a of [3]

Due to efforts by intel programmers, since Linux Kernel 5.10 [4, Installation Instructinos], there exists a driver for the DSA [5] which has no counterpart in the Windows OS-Family [4, Installation Instructinos], meaning code developed without an alternative path will not work there. To interface with the driver and perform configuration operations, intels libaccel-conf [6] user space toolset may be used which provides a command-line interface and can read configuration files to set up the device as described previously. After successful configuration, each WQ is exposed as a character device by `mmap` of the associated portal [3, p. 3].

Given the file permissions, it would now be possible for a process to submit work to the DSA via either MOVDIR64B or ENQCMD instructions, providing the descriptors by manually configuring them. This, however, is quite cumbersome, which is why Intels Data Mover Library [4] exists. With some limitations (like lacking support for DWQs) this library presents a high-level interface that takes care of creation and submission of descriptors, some error handling and reporting. Thanks to the high-level-view the code may choose a different execution path at runtime which allows the memory operations to either be executed in hardware (on a DSA) or in software (using equivalent instructions provided by the library) which makes code based upon it automatically compatible with systems that do not provide hardware or software support.

- drain descriptor / drain command signals completion of preceding descriptors for fencing in non-batch submissions, in batches the “fence flag“ can be used to ensure ordering, failures before a fence will lead to the following descriptors being aborted [1, p. 30], `sfence` or `mfence` should be executed before pushing drain descriptor [1, p. 32]
- cache control flag in descriptor controls whether writes are directed to cache or to memory [1, p. 31] effects on copy from DRAM > HBM unknown

2.2 Setup and Configuration

Give the reader the tools to replicate the setup. Also explain why the BIOS-configs are required.

Setup Requirements:

- VT-d enabled
- limit CPUPA to 46 Bits disabled
- IOMMU enabled
- kernel with iommu and idxd driver support
- kernel option ”intel_iommu=on,sm_on”

2.3 Programming Interface

- choice is intel data mover library
- two concepts, state-based for c-api and operation-based c++
- just explain the basics (no code) and refer to dml documentation

3 Performance Microbenchmarks

-

3.1 Benchmarking Methodology

-

3.2 Submission Method

- submit cost analysis: best method and for a subset the point at which submit cost $<$ time savings
- display the full opt-submitmethod graph
- maybe remeassure with higher amount of small copies? results look somewhat weird for 1k and 4k
- display the stacked bar of submit and complete time for single@1k, single@4k, single@1mib for HW-path and SW-path
- display the stacked bar of submit and complete time for batch50@1k, batch50@4k, batch50@1mib for HW-path and SW-path
- show batch because we care about the minimum task set size for a single producer (multi submit would be used for different task sets)
- conclude at which point using the DSA makes sense

3.3 Multithreaded Submission

- effect of mt-submit, low because SWQ implicitly synchronized, bandwidth is shared
- show results for all available core counts
- only display the 1engine tests
- show combined total throughput
- conclude that due to the implicit synchronization the sync-cost also affects 1t and therefore it makes no difference, bandwidth is shared, no guarantees on fairness

3.4 Multiple Engines in a Group

- assumed from arch spec that multiple engines lead to greater Performance
- reason is that page faults and access latency will be overlapped with preparing the next operation
- in the given scenario we observe the opposite, slight performance decrease
- show multisubmit 50 for both 1e and 4e
- maybe remeassure with each submission accessing different memory region?
- conclusion?

3.5 Data Movement from DDR to HBM

- present two copy methods: smart and brute force
- show graph for ddr->hbm intranode, ddr->hbm intrasocket, ddr->hbm intersocket
- conclude which option makes more sense (smart)
- because 4x or 2x utilization for only 1.5x or 1.25x speedup respectively
- maybe benchmark smart-copy intersocket in parallel with two smart-copies intra-socket VS. the same task with brute force

3.6 Analysis

- summarize the conclusions and define the point at which dsa makes sense
- minimum transfer size for batch/nonbatch operation
- effect of mtsubmit -> no fairness guarantees
- usage of multiple engines -> no effect
- smart copy method as the middle-ground between peak throughput and utilization
- lower utilization of dsa is good when it will be shared between threads/processes

4 Design

4.1 Detailed Task Description

- give slightly more detailed task Description
- perspective of ”what problems have to be solved”
- not ”what is querry driven prefetching”

4.2 Cache Design

The task of prefetching is somewhat aligned with that of a cache. As a cache is more generic and allows use beyond Query Driven Prefetching, the choice was made to solve the prefetching offload by implementing an offloading `Cache`. When refering to the provided implementation, `Cache` will be used from now on. The interface with `Cache` must provide three basic functions: requesting a memory block to be cached, accessing a cached memory block and synchronizing cache with the source memory. The latter operation comes in to play when the data that is cached may also be modified, requiring the entry to be updated with the source or the other way around. Due to the many possible setups and use cases, the user should also be responsible for choosing cache placement and the copy method. As re-caching is resource intensive, data should remain in the cache for as long as possible while being removed when system memory pressure due to restrictive memory size drives the `Cache` to flush unused entries.

4.2.1 Interface

To allow rapid integration and ease developer workload, a simple interface was chosen. As this work primarily focuses on caching static data, the choice was made only to provide cache invalidation and not synchronization. Given a memory address, `Cache::Invalidate` will remove all entries for it. The other two operations are provided in one single function, which we shall call `Cache::Access` henceforth, receiving a data pointer and size it takes care of either submitting a caching operation if the pointer received is not yet cached or returning the cache entry if it is. The cache placement and assignment of the task to accelerators are controlled by the user. In addition to the two basic operations outlined before, the user also is given the option to flush the cache using `Cache::Flush` of unused elements manually or to clear it completely with `Cache::Clear`.

As caching is performed asynchronously, the user may wish to wait on the operation. This would be beneficial if there are other threads making progress in parallel while the current thread waits on its data becoming available in the faster cache, speeding up local computation. To achieve this, the `Cache::Access` will return an instance of an object which from hereinafter will be refered to as `CacheData`. Through `CacheData::GetDataLocation` a pointer to the cached data will be retrieved, while also providing `CacheData::WaitOnCompletion` which must only return when the caching operation has completed and during which the current thread is put to sleep, allowing other threads to progress.

4.2.2 Cache Entry Reuse

When multiple consumers wish to access the same memory block through the `Cache`, we could either provide each with their own entry, or share one entry for all consumers. The first option may cause high load on the accelerator due to multiple copy operations being submited and also increases the memory footprint of the system. The latter option requires synchronization and more complex design. As the cache size is restrictive, the latter was chosen. The already existing `CacheData` will be extended in scope to handle this by allowing copies of it to be created which must synchronize with each other for `CacheData::WaitOnCompletion` and `CacheData::GetDataLocation`.

4.2.3 Cache Entry Lifetime

By allowing multiple references to the same entry, memory management becomes a concern. Freeing the allocated block must only take place when all copies of a `CacheData` instance are destroyed, therefore tying cache entry lifetime to the lifetime of the longest living copy of the original instance. This makes access to the entry legal during the lifetime of any `CacheData` instance, while also guaranteeing that `Cache::Clear` will not have any unforseen side effects, as deallocation only takes place when the last consumer has `CacheData` go out of scope or manually deletes it.

4.2.4 Usage Restrictions

As cache invalidation applies mainly to non-static data which this work does not focus on, two restrictions are placed on the invalidation operation. This permits drastically simpler cache design, as a fully coherent cache would require developing a thread safe coherence scheme which is outside our scope.

Firstly, overlapping areas in the cache will cause undefined behaviour during invalidation of any one of them. Only the entries with the equivalent source data pointer will be invalidated, while other entries with differing source pointers which, due to their size, still cover the now invalidated region, will not be invalidated and therefore the cache may and may continue to contain invalid elements at this point.

Secondly, invalidation is to be performed manually, requiring the programmer to remember which points of data are at any given point in time cached and invalidating them upon modification. No ordering guarantees will be given for this situation, possibly leading to threads still having a pointer to now-outdated entries and continuing their progress with this.

Due to its reliance on libnuma for numa awareness, `Cache` will only work on systems where this library is present, excluding, most notably, Windows from the compatibility list.

4.2.5 Thread Safety Guarantees

After initialization, all available operations for `Cache` and `CacheData` are fully threadsafe but may use locks internally to achieve this. In 5 we will go into more detail on how these guarantees are provided and how to optimize the cache for specific use cases that may warrant less restrictive locking.

4.2.6 Accelerator Usage

Compared with the challenges of ensuring correct entry lifetime and thread safety, the application of DSA for the task of duplicating data is simple, thanks partly to Intel Data Mover Library (Intel DML) [4]. Upon a call to `Cache::Access` and determining that the given memory pointer is not present in cache, work will be submitted to the Accelerator. Before, however, the desired location must be determined which the user-defined cache placement policy function handles. With the desired placement obtained, the copy policy function then determines, which nodes should take part in the copy operation which is equivalent to selecting the Accelerators following 2.1.1. This causes the work to be split upon the available accelerators to which the work descriptors are submitted at this time. The handlers that Intel DML [4] provides will then be moved to the `CacheData` instance to permit the callee to wait upon caching completion. As the choice of cache placement and copy policy is user-defined, one possibility will be discussed in 5.

5 Implementation

...implementation ...

6 Evaluation

...evaluation ...

7 Conclusion And Outlook

...conclusion ...

Glossary

A

ATC

... desc ...

B

BAR

... desc ...

D

DMR

... desc ...

DSA

... desc ...

DWQ

... desc ...

E

Engine

... desc ...

ENQCMD

... desc ...

G

Group

... desc ...

I

Intel DML

... desc ...

IOMMU

... desc ...

M

MOVDIR64B

... desc ...

P

PASID

... desc ...

S

SWQ

... desc ...

W

WQ

... desc ...

Bibliography

[1] Intel. ‘Intel® Data Streaming Accelerator Architecture Specification’. (16th Sep. 2022), [Online]. Available: https://www.intel.com/content/www/us/en/content-details/671116/intel-data-streaming-accelerator-architecture-specification.html (visited on 15th Nov. 2023).

[2] Intel. ‘New Intel® Xeon® Platform Includes Built-In Accelerators for Encryption, Compression, and Data Movement’. (Dec. 2022), [Online]. Available: https://www.intel.com/content/dam/www/central-libraries/us/en/documents/2022-12/storage-engines-4th-gen-xeon-brief.pdf (visited on 15th Nov. 2023).

[3] R. K. et al. ‘A Quantitative Analysis and Guideline of Data Streaming Accelerator in Intel® 4th Gen Xeon® Scalable Processors’. (May 2023), [Online]. Available: https://arxiv.org/pdf/2305.02480.pdf (visited on 7th Jan. 2024).

[4] Intel, *Intel Data Mover Library Documentation*, https://intel.github.io/DML/index.html. (visited on 7th Jan. 2024).

[5] Intel, *Intel IDXD Driver for Linux Kernel*, https://github.com/intel/idxd-driver. (visited on 7th Jan. 2024).

[6] Intel, *Intel IDXD User Space Application*, https://github.com/intel/idxd-config. (visited on 7th Jan. 2024).