Exploring the Limits of Cross-Platform Sparse Tensor Processing

Abstract

Sparse tensors are the natural way to store and represent multi-dimensional data, but ensuring their efficient processing is an important open challenge. Some tensor methods, like Matricised Tensor Times Khatri-Rao Product (MTTKRP) and Tensor Times Matrix (TTM), pose as major performance bottlenecks for commonly used algorithms in many research areas, especially when considering that most real-world data is highly sparse. State-of-the-art optimisations tend to focus on single-device vendor-specific implementations, which are not applicable to modern heterogeneous systems with several compute devices of different architectures, such as multi-core CPUs, GPUs and FPGAs. To close this gap, in this work, novel portable and highly data-parallel software approaches and specialized FPGA architectures are proposed for the most prominent sparse tensor methods (TTM and MTTKRP), which allow for their efficient cross-platform SYCL-based processing in modern heterogeneous systems at different granularity levels. We also conduct an in-depth analytical and experimental characterization of the processing upper-bounds that these sparse tensor methods can achieve in multi-core CPU, GPU, heterogeneous and FPGA-based processing platforms from different vendors, achieving speedups of up to 6× for TTM and 7× for MTTKRP, when compared to the state-of-the-art approaches, and with the advantage of not being device nor vendor specific.

***Index Terms*—Sparse Tensors, TTM, MTTKRP, SYCL, Heterogeneous Systems, Field Programmable Gate Array (FPGA)**

1. Introduction

With the exponential increase in data consumption worldwide, most major research areas are applying complex algorithms to compute and extract information. A diverse range of scientific areas such as healthcare [1, 2], machine learning [3, 4], quantum chemistry [5, 6], social network analytics [7], deep learning [8, 9] and cybersecurity [10] show how spread the need is. However, data retrieved from the real-world is commonly sparse and multi-dimensional, therefore, attaining its efficient processing is far from a trivial task [11]. Tensors, which are multi-way arrays, provide a logical way to represent this multi-dimensional data. Thus, the increased interest in optimising sparse tensor computations. For example, modern approaches in artificial intelligence, and neural networks, in specific the ones used for deep learning, typically operate on large multidimensional structures, e.g. images and videos [12]. To mitigate the high memory and computing demands when processing a large number of layers and parameters [13], network [14] is typically applied, which introduces sparsity in the computations at no significant accuracy loss.

Real-world tensors typically have huge storage requirements, with some can take up to petabytes [15]. However, as aforementioned, they are also highly sparse. The sparsity of these datasets impacts the performance when processing them. Retrieving elements displaced throughout the tensor diminishes possible reuse of the data, increasing the strain on the memory. To face this challenge, a plethora of sparse tensor storage formats are proposed, which aim at efficient processing with reduced memory footprint [16, 17]. Some of the more common storage formats are the Coordinate (COO) format as well as the Compressed Sparse Fiber (CSF) format [18].

Due to the large number of terms, processing, storing, interpreting, or extracting patterns from a tensor can be a challenge. Therefore, a common analysis of the tensor takes the form of decomposing it into interpretable components [19]. The two most prominent methods for tensor decomposition are Tucker Decomposition [20, 21] and Canonical Polyadic Decomposition (CPD) [11, 22]. The first method decomposes a tensor into a core tensor and multiple matrices, which correspond to different core scalings along each mode. The second expresses a tensor as a sum of rank one tensors. Both of these algorithms have a tensor method as their performance bottleneck, for Tucker Decomposition it is Tensor Times Matrix (TTM) and for CPD it is Matricised Tensor Times Khatri-Rao Product (MTTKRP).

To reach unprecedented performance levels, contemporary computing platforms rely on high level of heterogeneity, by combining the devices of different architectures into a single execution environment. At almost every scale, from embedded systems up to supercomputing environments, modern computing platforms embrace multi-core Central Processing Units (CPUs) and Graphics Processing Units (GPUs) devices, with an evident rising trend in the use of specialized architectures deployed in the FPGA fabric on the road to pursuing energy-efficiency. As such, these heterogeneous platforms offer a tremendous computing power to address the challenges of efficient sparse tensor computing. However, their capabilities for this kind of processing are yet to be fully exploited.

From the vast amount of available architectures it is non-trivial to pinpoint the characteristics and capabilities that best suite the specific needs of sparse tensor computing. State-of-the-art approaches, such as SPLATT [23] and ParTI [24], predominately focus on providing single-device vendor-specific implementations for sparse tensor computing, by developing complex algorithms and storage formats to fit the hardware characteristics of the targeted processing device. However, with the emerging heterogeneous systems, there

is an opportunity to exploit the variety of computational architectures available. Recent attempts on heterogeneous processing [25, 26] resort to vendor-specific frameworks, hence limiting the devices that can be targeted.

A typical burden in employing the devices of different architectures for efficient cross-device processing revolves around the platform programability, efficiently exploiting the capabilities of each device architecture and difficulty in unifying different device-specific programming models, frameworks and tools into a single processing environment. For example, OpenMP and vector intrinsics for the CPU, CUDA and HIP for the GPU and Verilog, VHDL and Xilinx's Vivado HLS for the FPGA. OpenCL is a notable attempt to tackle the heterogeneous processing, but, even after more than a decade of developments, its wide adoption is yet to be evidenced, which is typically restrained by its low level nature, limited adoption by major vendors and inability to match the performance of hand-tuned codes [27]. SYCL, however, is attracting attention precisely for the simplicity offered when interacting with heterogeneous systems. Modern systems have access to different kinds of accelerators, with SYCL offloading computation performed independently of which accelerator is being targeted, enabling the possibility of utilising a single source code in order to target multiple accelerators. Such portability allied with performance on par with the current standards [28], makes SYCL a potential standard in heterogeneous computing.

The goal of this work is to explore the processing upper-bounds for sparse tensor computing on modern CPU, GPU and FPGA systems. With this aim, we propose a set of portable, cross-platform and optimised SYCL-based approaches for the most prominent sparse tensor methods, i.e., TTM and MTTKRP, which allow for efficient data-parallel and heterogeneous processing at different levels of granularity. While these approaches allow to experimentally uncover the performance limits of sparse tensor processing for different architectures, performance modelling is also applied to obtain these limits analytically. The key findings and contributions of this paper are summarized as follows:

- Novel SYCL-based and optimised TTM and MTTKRP approaches for programmable architectures, namely CPU and GPU;
- Analytical and experimental methodology for uncovering the limits of sparse tensor processing on CPU and GPU, based on a set of specifically designed synthetic sparse tensors;
- SYCL-driven specialized architecture design for TTM and MTTKRP, deployed in the FPGA fabric;
- The first SYCL heterogeneous (CPU+GPU) framework for sparse tensor computations;
- Extensive experimental campaign on devices from different vendors showcasing that the proposed approach outcomes the state-of-the-art solutions by up to 5× for TTM, and 6× for MTTKRP.

TABLE 1: Notation used throughout the study

SlcCnt	total number of slices in the tensor
FbrCnt	total number of fibers in the tensor
NnzCnt	total number of non-zero elements in the tensor
ColCnt	total number of columns in the matrices
FbrPSlc	number of fibers in a slice
NnzPSlc	number of non-zero elements in a slice
NnzPFbr	number of non-zero elements in a fiber

2. Background and Related Work

Tensors are multidimensional arrays which represent high dimensional data. A tensor's order is the dimensionality of the array, e.g. a first-order tensor is a vector while a second-order tensor is a matrix. Tensors of order greater or equal than three are typically referred to as high-order tensors. For simplicity, it is common to apply the term tensor only to high-order tensors, while matrices and vectors are referred to as low-order tensors [29]. Dimensions are typically referred to as modes, so a tensor has as many modes as its order, e.g. a third-order tensor has mode-zero, mode-one and mode-two. A fiber is a vector formed by fixing all modes of the tensor but one. As an example, the fibers of a matrix are either its rows or its columns depending on the mode fixed. Similarly, a slice is a matrix formed by fixing all modes of the tensor but two.

High-order tensors have huge storage requirements since they tend to have several millions of elements. In sparse tensors, this issue becomes even more prevalent, since zeros need not to be stored nor computed upon. Therefore, state-of-the-art storage formats for sparse tensors are optimised to minimise these requirements by, amongst other techniques, not storing the zeros, hence also reducing the number of memory accesses during computation. One of the most widely used formats, and the one used for the development of our implementations, is the CSF format [18]. This format is based on the idea of a tree like structure, where each mode is a level and paths from root to leaf encode a nonzero coordinate.

Furthermore, let us establish Table 1 as the notation for the remainder of this study.

2.1. Tensor Methods

Two of the most prominent tensor methods/operations are TTM and MTTKRP. The first belongs to the category of Tensor Contractions (TCs), which are a generalisation of General Matrix Multiplication (GEMM), while the latter belongs to the category of Sequence Operations. The reason for the relevance of these algorithms is their use as key part in Tensor Decomposition, whose purpose is to approximate a high-order tensor with lower-order tensors.

TTM is the analogue to GEMM, but with a K-th order tensor instead of a second order one. Like in GEMM, all fibers are multiplied against the matrix, hence consisting of several vector-matrix dot products. Let us denote tensor $T \in \mathbb{R}^{I \times J \times K}$ and matrix $M \in \mathbb{R}^{K \times F}$ as input, with the output being tensor $O \in \mathbb{R}^{I \times J \times F}$, then the TTM between them can be expressed as follows:

$$O_{ijf} = \sum_{k} \left(T_{ijk} \times M_{kf} \right) \tag{1}$$

MTTKRP uses, as inputs, a K-th order tensor and K-1 matrices, and outputs a matrix. The tensor is matricised and the matrices are used to compute a Khatri-Rao product. The Khatri-Rao product is a column-wise Kronecker product. The Kronecker product is a generalization of a matrix outer-product. Given two matrices $A \in \mathbb{R}^{I \times J}$ and $B \in \mathbb{R}^{K \times R}$ their Kronecker product is denoted by matrix $C \in \mathbb{R}^{IK \times JR}$,

$$C = A \otimes B = \begin{bmatrix} a_{00}B & \dots & a_{0J}B \\ \vdots & & \vdots \\ a_{I0}B & \dots & a_{IJ}B \end{bmatrix} \tag{2}$$

Given two matrices $A \in \mathbb{R}^{I \times R}$ and $B \in \mathbb{R}^{J \times R}$ their Khatri-Rao product is denoted by matrix $C \in \mathbb{R}^{IJ \times R}$,

$$C = A \odot B = \begin{bmatrix} a_{:0} \otimes b_{:0} & \dots & a_{:R} \otimes b_{:R} \end{bmatrix} \tag{3}$$

Matricisation is a kind of reshaping that flattens all tensor's modes but one, hence changing the tensor to a second-order one, in other words, a matrix. MTTKRP consists of a GEMM between the matricised tensor and the Khatri-Rao product of $K-1$ matrices with same number of columns, where K is the order of the tensor before matricising. Given $K-1$ matrices with R columns the Khatri-Rao product of these is a matrix $U \in \mathbb{R}^{M_0 \dots M_{K-2} \times R}$ and the output of the MTTKRP is a matrix $V \in \mathbb{R}^{M_{K-1} \times R}$.

For sparse MTTKRP, from the mode indexes of each nonzero element it is possible to extract the row of the output the element contributes to and the rows of the matrices that the element multiplies with. Since it is possible to compute each element's individual contribution to the output without having to compute neither the matricised tensor nor the Khatri-Rao products, in order to avoid redundant computation and extra storage, these formulations tend to not be implemented directly but rather integrated into tensor operations.

State-of-the-art implementations tend to focus on one of two different approaches. ParTI [24] distributes the non-zero elements across the threads for processing, thus ensuring load-balance. However, this approach then requires synchronisation to reduce each thread's contribution. SPLATT [23], on the other hand, does the workload distribution, for MTTKRP, slice-wise, hence avoiding atomics and locks. We follow a similar strategy to the one used by SPLATT, however we expand our approach to work on more devices from multiple vendors.

2.2. Heterogeneous Programming Model

For this study, we target three different architectures: CPU, GPU and FPGA. The first, being designed to increase Instruction-Level Parallelism (ILP), is better suited for coarse-grained parallelism, which also makes better use of the cache hierarchy present in the architecture [30]. The GPU, with its massively parallel architecture, is instead better suited for fine-grained parallelism through multi-threaded execution [31]. Lastly, the FPGA, being a re-configurable device [32], exploits spatial parallelism, reducing the control overhead an improving the efficiently usage of the hardware resources.

To exploit the full capabilities of these architectures, commonly available in modern heterogeneous systems, a developer is forced to be familiar with a wide spectrum of APIs, frameworks and languages. For example, OpenMP [33] and vector intrinsics for the CPU, CUDA and HIP for the GPU or Verilog, VHDL and Xillinx's Vivado HLS for the FPGA. To avoid this burden, a unifying programming model that allows the development of cross-platform code is required. Khronos SYCL provides the desired model, ensuring both portability and comparable performance to other solutions [28, 34], so much that it can be considered a future standard for heterogeneous computing.

SYCL defines an abstract Single Program Multiple Data (SPMD) programming model where developers program at a higher level than the native acceleration Application Programming Interface (API), but always have access to lower-level code that enables having a single-source code targeting any accelerator. However, in order for a developer to take the most performance out of an accelerator, the code must be adapted to the susceptibilities of that accelerator's architecture. There are many SYCL implementations, with Intel's OneAPI DPC++ being one of the most actively developed open-source implementations.

2.3. Performance Modelling with CARM

Cache Aware Roofline Model (CARM) is an insightful architecture performance model, which provides an intuitive way of visually representing the limits of the performance of the architecture, considering the computational cores and the memory system [35]. From an architecture perspective, the overall execution of a program can be mainly limited either by the processor computation capabilities or by the capabilities of the memory subsystem. By relying on this observation, CARM models the performance upper-bounds for a given architecture in respect to the amount of performed computations over the amount of requested data (bytes), i.e., Arithmetic Intensity (AI). CARM explicitly considers all levels of memory hierarchy by observing the complete amount of requested data. The AI and performance can be differently expressed depending on the number of operations performed (N_{ops}) and the number of bytes accessed in the memory. Since all data types used in this work are 4-byte wide, the AI can be expressed by (4).

$$\text{AI} = \frac{1}{4} \times \frac{N_{ops}}{N_{loads} + N_{stores}} \tag{4}$$

3. Exploring Sparse Tensor Processing on General-Purpose Architectures

We propose herein a SYCL-based approach for sparse tensor TTM and MTTKRP processing based on the CSF

TABLE 2: Arithmetic Intensity (AI) on Programmable Architectures

	Element-Centric TTM	Fiber-Centric TTM	Element-Centric MTTKRP	Row-Centric MTTKRP
N_{ops}	$2 \times NnzPFbr$	$2 \times ColCnt \times NnzPFbr$	$2 \times (FbrPSlc + NnzPSlc)$	$2 \times ColCnt \times (FbrPSlc + NnzPSlc)$
N_{loads}	$2 + 3 \times NnzPFbr$	$2 + NnzPFbr \times (ColCnt + 2)$	$3 \times (1 + NnzPSlc + FbrPSlc)$	$(ColCnt + 2) \times (FbrPSlc + NnzPSlc) + 3$
N_{stores}	1	$ColCnt$	1	$ColCnt$
AI_{min}	1/12	1/12	1/10	1/10
AI_{max}	1/6	1/2	1/6	1/2

storage format. This format, when compared to COO, tends to require less memory accesses by condensing shared indexes, hence delivering more performance for methods that are typically memory bound. To allow for an in-depth assessment of the performance upper-bounds for parallel sparse tensor computing on CPU and GPU architectures, we tackle two different levels of processing granularity, i.e., element-centric and fiber-centric sparse tensor execution.

3.1. Tensor Times Matrix

The proposed Element-Centric TTM processing assigns each thread to compute one element of the output. Therefore, each thread requires access to one fiber and to one column of the matrix. This approach attempts to maximise parallelism through a 2D kernel that generates one thread for each independent task, thus avoiding synchronisation between threads.

```
1  range<2> globalSize(fbrCnt, colCnt);
2  range<2> localSize(1, colCnt);
3  nd_range<2> numItems(globalSize, localSize);
4
5  event e { q.submit([&](handler &h) {
6      h.parallel_for(numItems,
7                     [=](nd_item<2> item) {
8          const auto fbr { item.get_global_id(0) };
9          const auto col { item.get_local_id(1) };
10         auto tmp { 0.0f };
11
12         for (auto ele { accFbrPtr[fbr] };
13              ele < accFbrPtr[fbr+1]; ++ele) {
14             const auto k {
15                 (accKIdx[ele]-1)*colCnt };
16             const auto val { accValues[ele] };
17             tmp += val * accMatrix[k+col];
18         }
19
20         accOutput[fbr*colCnt+col] = tmp;
21     });
22 }) };
```

Listing 1: Element-Centric TTM

Listing 1 presents the SYCL kernel of the proposed approach. Every thread starts by loading the boundaries of its assigned fiber (lines 12-13). Then for each non-zero element in that fiber, the threads loads its index and value (lines 14-16). The index specifies which element of the assigned column is to be loaded, while the value is computed against that same element (line 17). When all elements in the fiber have been computed, the accumulated result is stored to global memory (line 20).

The AI of each thread, depicted on Table 2, may differ depending on *NnzPFbr*. However, it will always range between a minimum and a maximum value, which can be calculated. The minimum can be achieved when the fiber has the least possible number of non-zero elements, which is one. Inversely, the maximum can be achieved when *NnzPFbr* is at its highest.

Another approach to efficiently extract data-parallelism in TTM processing is to assign each thread to compute an entire fiber of the output (instead of a single element). In this approach, each thread still requires access to one fiber, but now it also requires access to the whole matrix instead of just a column (as previously elaborated in Element-Centric TTM). We refer to this approach as Fiber-Centric TTM.

```
1  range<1> globalSize(fbrCnt);
2  range<1> localSize(wgSize);
3  nd_range<1> numItems(globalSize, localSize);
4
5  event e { q.submit([&](handler &h) {
6      h.parallel_for(numItems,
7                     [=](nd_item<1> item) {
8          const auto fbr { item.get_global_id(0) };
9          float tmp[colCnt];
10
11         for (auto col { 0 }; col < colCnt; ++col){
12             tmp[col] = 0.0f;
13         }
14
15         for (auto ele { accFbrPtr[fbr] };
16              ele < accFbrPtr[fbr+1]; ++ele) {
17             const auto k {
18                 (accKIdx[ele]-1)*colCnt };
19             const auto val { accValues[ele] };
20
21             for (auto col { 0 }; col < colCnt;
22                  ++col) {
23                 tmp[col] += val*accMatrix[k+col];
24             }
25         }
26
27         for (auto col { 0 }; col < colCnt; ++col){
28             accOutput[fbr*colCnt+col] = tmp[col];
29         }
30     });
31 }) };
```

Listing 2: Fiber-Centric TTM

The SYCL kernel presented in Listing 2 is based on a 1D kernel. This approach enforces consecutive accesses to the matrix's row at the expense of granularity. Every thread also starts by loading its fiber boundaries (lines 15-16) as well as the index and value for each of the non-zero elements within those boundaries (lines 17-19). Then, each of the indexes specifies one of the matrix's rows. That row is loaded and computed against the value of that same non-zero element (lines 21-24). When all elements in the

fiber have been computed, the accumulated results for all columns are stored to global memory (lines 27-29).

Each thread's AI, Table 2, is now not only dependent on *NnzPFbr* but also on *ColCnt*. Therefore, the minimum AI can be achieved when *NnzPFbr* and *ColCnt* are both, simultaneously, at their lowest. While the maximum can be achieved when they are both at their highest.

3.2. Matricised Tensor Times Khatri-Rao Product

For the proposed Element-Centric MTTKRP, the approach is analogue to the one presented for Element-Centric TTM processing. Each thread computes one element of the output, therefore each thread is assigned with one slice of the tensor, one column of matrix1 and one column of matrix2. Note that these columns must match, e.g. a thread gets column zero on both matrices. We call this approach Element-Centric MTTKRP.

```
1  range<2> globalSize(slcCnt, colCnt);
2  range<2> localSize(1, colCnt);
3  nd_range<2> num_items(globalSize, localSize);
4
5  event e { q.submit([&](handler &h) {
6      h.parallel_for(num_items,
7                     [=](nd_item<2> item) {
8          const auto slc { item.get_global_id(0) };
9          const auto col { item.get_local_id(1) };
10         auto inB { 0.0f }, inC { 0.0f };
11
12         for (auto fbr { accSlcPtr[slc] };
13              fbr < accSlcPtr[slc+1]; ++fbr) {
14             const auto j {
15                 (accFbrIdx[fbr]-1)*colCnt };
16
17             for (auto ele { accFbrPtr[fbr] };
18                  ele < accFbrPtr[fbr+1]; ++ele){
19                 const auto k {
20                     (accKIdx[ele]-1)*colCnt };
21                 const auto val { accValues[ele] };
22                 inC += val*accMatrix2[k+col];
23             }
24
25             inB += inC*accMatrix1[j+col];
26             inC = 0.0f;
27         }
28
29         accOutput[slc*colCnt+col] = inB;
30     });
31 }) };
```

Listing 3: Element-Centric MTTKRP

In Listing 3, we present a 2D Kernel that aims at exploiting parallelism by assigning each thread one independent computation. Each thread starts by loading the slice boundaries (lines 12-13) followed by the fiber's boundaries and index for each fiber within the assigned slice (lines 14-18). For each non-zero element on the current fiber, both the element's index and value are loaded (lines 19-21). This is repeated for every fiber in the slice, hence being done for every single non-zero element in the slice. From the assigned column in the matrices, the fiber and element indexes specify, respectively, the elements to be loaded from matrix1 and matrix2. Each non-zero element is computed against the corresponding element of matrix2's column (line 22). The accumulated result of the non-zero elements in a fiber are computed against that fiber's corresponding element of matrix1's column (line 25). After all fibers in the slice have been computed, their accumulated result is stored to global memory (line 29).

From Table 2, it is possible to observe that a thread's AI is dependent on *FbrPSlc* and *NnzPSlc*. The minimum AI attainable by a thread happens when *NnzPSlc* is the smallest it can be, meaning one. This forces *FbrPSlc* to also be one as the only fibers with at least one non-zero element are stored. On the other hand, the maximum achievable AI is attained when (*FbrPSlc* + *NnzPSlc*) is large enough to mitigate the number of loads and stores.

```
1  range<1> globalSize(slcCnt);
2  range<1> localSize(wgSize);
3  nd_range<1> num_items(globalSize, localSize);
4
5  event e { q.submit([&](handler &h) {
6      h.parallel_for(num_items,
7                     [=](nd_item<1> item) {
8          const auto slc { item.get_global_id(0) };
9          float inB[colCnt], inC[colCnt];
10
11         for (auto col { 0 }; col < colCnt; ++col){
12             inB[col] = inC[col] = 0.0f;
13         }
14
15         for (auto fbr { accSlcPtr[slc] };
16              fbr < accSlcPtr[slc+1]; ++fbr) {
17             const auto j {
18                 (accFbrIdx[fbr]-1)*colCnt };
19
20             for (auto ele { accFbrPtr[fbr] };
21                  ele < accFbrPtr[fbr+1]; ++ele){
22                 const auto k {
23                     (accKIdx[ele]-1)*colCnt };
24                 const auto val { accValues[ele] };
25
26                 for (auto col { 0 }; col < colCnt;
27                      ++col) {
28                     inC[col] += (
29                         val*accMatrix2[k+col] );
30                 }
31             }
32
33             for (auto col { 0 }; col < colCnt;
34                  ++col) {
35                 inB[col] += (
36                     inC[col]*accMatrix1[j+col] );
37                 inC[col] = 0.0f;
38             }
39         }
40
41         for (auto col { 0 }; col < colCnt; ++col){
42             accOutput[slc*colCnt+col] = inB[col];
43         }
44     });
45 }) };
```

Listing 4: Row-Centric MTTKRP

Data-parallelism can also be efficiently extracted in MTTKRP processing by assigning to each thread the entirety of both matrices. This makes each thread responsible for the computation of a whole output row, instead of a single element as happened in Element-Centric MTTKRP. We call this approach Row-Centric MTTKRP.

The differences between the approach in Listing 4 and the Element-Centric one lie in the loads from the matrices, the stores and the number of operations. Our Row-Centric approach creates a 1D Kernel with as many threads as there are slices in the tensor. For each fiber in the assigned slice, one row of matrix1 is loaded (lines 33-38), which leads to *ColCnt* Multiply-Accumulates (MACs) per fiber. Similarly, for each non-zero element, one row of matrix2 is loaded (lines 26-30), which leads to *ColCnt* MACs per element. Finally, each thread computes one row of the output (lines 41-43), therefore the number of stores is *ColCnt*.

As presented in Table 2, the AI now also depends on *ColCnt*. To minimise the AI, *ColCnt* and (*FbrPSlc*+*NnzPSlc*) must be at their minimum. The minimum *ColCnt* is one and reflects the scenario where the matrices are actually vectors. For *NnzPSlc*, it is also one and enforces *FbrPSlc*=1, since only fibers with at least one non-zero element are stored. Inversely, maximising the AI requires *ColCnt* and (*FbrPSlc* + *NnzPSlc*) to be as large as possible.

3.3. Heterogenous Approach

To develop an heterogeneous implementation for TTM and MTTKRP, since we already have two kernels for each method, the greatest challenge is to decide on how to split the workload across the targeted architectures. The first step is the creation of multiple SYCL queues, one for each device, as can be observed in Listing 5. Then to decide on the strategy for the workload distribution. We developed two different strategies: one is a pure static distribution, the other is what we call an adaptive static distribution.

```
1 std::vector<device> D{ device(cpu_selector_v),
2                        device(gpu_selector_v) };
3
4 std::vector<queue> Q;
5 for (auto d : D) Q.push_back(queue(d));
```

Listing 5: Queue creation for Heterogeneous Approach

For the purely static approach, we submit the corresponding workload to each one of the queues, with the workload being the the fibers to process for TTM or the slices for MTTKRP. This approach may lack load balance, if the number of non-zero elements in the fibers or the number of fibers in the slices differ. Also, without prior knowledge over the architectures present in the system, the proportion in which the workload is distributed may be inadequate for the devices' computational capabilities.

The second starts by splitting the workload in two portions, with the workload, once again, being either the fibers or the slices depending on the method. One portion consists of 10% of the data. This smaller portion is divided in equal parts and processed by the devices. When all devices are done, the execution time of each is measured and used to define the adequate proportion the distributing the remaining workload. The other portion, consisting of 90% of the data, is then split according to the derived proportion. While the adaptive approach allows for better workload distribution in general, it requires the first 10% of the data to be processed serially in relation to the remaining 90%. When compared to the purely static approach, this tends to be better unless the static distribution is already very close to the one derived by the adaptive approach.

4. Exploring Sparse Tensor Processing on Specialised Architectures

Unlike the general-purpose architectures, the design deployed in the FPGA is not fixed. It is instead defined depending on the target algorithm, therefore the analysis of these specialized architectures requires different methodology. It consists of theoretically computing both the AI and peak performance of the implemented designs. By relying on this analysis and the CARM concepts, it is expected to confirm the bottlenecks and test the utilization limits.

On the FPGA, typically the operations performed by the Digital Signal Processings (DSPs) are considered to determine the Peak Performance. Depending on the design and operation being executed, the operating frequency and percentage of available resources for the actual design may change, hence soft-logic is not considered for the sake of a fair comparison. Throughout this study, we assume the resource limiting the number of Processing Elements (PEs) is the amount of DSPs available on the FPGA.

4.1. Tensor Times Matrix

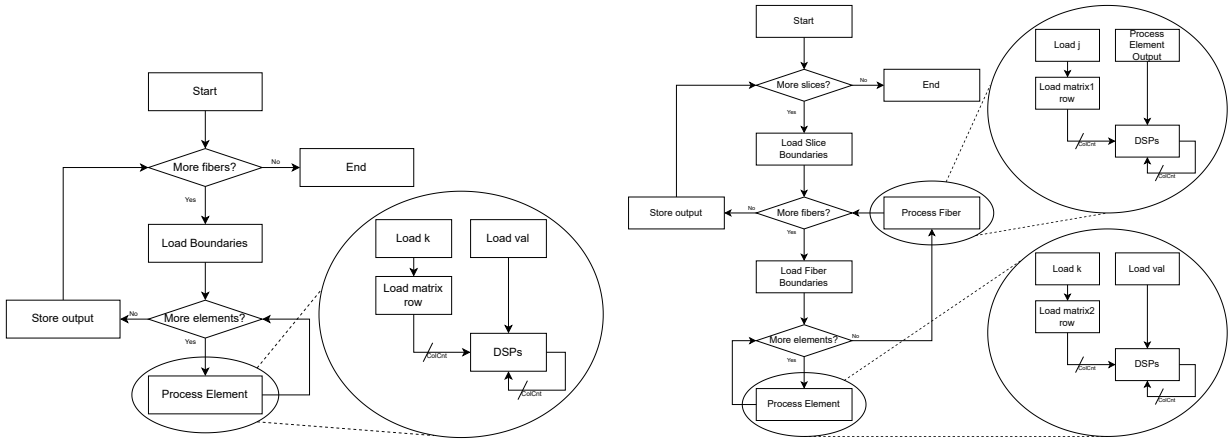
Our design for TTM processing on the FPGA is depicted in Figure 1a, which is organized in a similar fashion as the previously elaborated kernels for general-purpose architectures. For each fiber in the tensor, it loads the fiber boundaries. Then, for each non-zero element in the fiber, it loads the index and value of that element. Finally, a row of the matrix is loaded and computed against the element. When all elements of the current fiber have been computed, the output fiber is stored to global memory.

The main bottleneck of this design is the number of loads, specifically the ones from the matrix. For each non-zero element, a full row is loaded. However, due to the absence of cache hierarchy, even if the same row is requested by one of the following elements, that row will always have to be loaded again. To overcome this challenge, we resorted to the FPGA on-chip memory. By loading the whole matrix to this faster memory before starting the computations, it is possible to reduce the average access time to data in the matrix. However, it comes with a trade-off, as matrices for real-world datasets tend to be very large, the amount of resources used is greater, which causes the design to either not fit in some FPGAs or to be constrained to a maximum amount of columns in the matrix. It is important to notice that adjusting the unroll factor, can also contribute to providing certian flexibility between resource utilisation and performance.

The AI of this design depends on the characteristics of the tensor as well as the number of columns in the matrix. The total amount of loads associated with the tensor are $FbrCnt + 1$ for the fiber boundaries and $2 \times NnzCnt$ for the non-zero indexes and values. The loads associated with the matrix are $NnzCnt \times ColCnt$, as one row is loaded per

TABLE 3: AI on Specialised Architectures

	TTM	**MTTKRP**
N_{ops}	$2 \times ColCnt \times NnzCnt$	$2 \times ColCnt \times (FbrCnt + NnzCnt)$
N_{loads}	$FbrCnt + NnzCnt \times (ColCnt + 2) + 1$	$SlcCnt + 2 + (ColCnt + 2) \times (FbrCnt + NnzCnt)$
N_{stores}	$FbrCnt \times ColCnt$	$SlcCnt \times ColCnt$
AI_{min}	1/12	1/10
AI_{max}	1/2	1/2



(a) TTM (b) MTTKRP

Figure 1: Diagram of Processing Element

non-zero element. There are $FbrCnt \times ColCnt$ stores and $NnzCnt \times ColCnt$ MACs, meaning twice as much operations.

From Table 3, it is possible to minimise and maximise the AI by changing the tensor's characteristics. To minimise it, one must decrease the number of operation while increasing the number of loads and stores. This means *NnzCnt* and *ColCnt* are at their minimum and *FbrCnt* is at its maximum. For the first two, the minimum is one, i.e., one non-zero element in the tensor and a matrix with one column (vector). As for the latter, since only fibers with at least one non-zero element are stored, the maximum is as many fibers as non-zero elements, which in this case is also one. Consequently, for the maximum, both *NnzCnt* and *ColCnt* must be as large as possible (to perform more operations), while *FbrCnt* must be as small as possible (to decrease the number of data movements), i.e., one fiber holding all non-zero elements. Finally, we analyse the maximum attainable performance by this Processing Element PE and how it compares to the total FPGA Peak Performance. This design uses one DSP for every column of the matrix, therefore one PE uses *ColCnt* DSPs and performs $2 \times ColCnt$ operations per cycle.

4.2. Matricised Tensor Times Khatri-Rao Product

Our design for MTTKRP processing on the FPGA is depicted on Figure 1b. Although introducing another layer of depth with the inclusion of slices, it is similar to the design presented for TTM processing. For each slice in the tensor, it loads the slice boundaries. Then for each fiber in the slice, it loads that fiber's boundaries and for each non-zero element within the fiber, it loads the index and value of that element. Next, a row of matrix2 is loaded and computed against the element. When all elements of the current fiber have been computed, the accumulated results of all elements are computed against a row of matrix1. Once the current slice has no more fibers to process, the output row is stored to global memory.

In this design, the AI depends on the tensor characteristics and the number of columns in the matrices. The total amount of loads associated with the tensor are $SlcCnt+1$ for the slice boundaries, $2 \times FbrCnt+1$ for the fiber boundaries and indexes and $2 \times NnzCnt$ for the non-zero indexes and values. The total amount of loads for the matrices are $FbrCnt \times ColCnt$ for matrix1 and $NnzCnt \times ColCnt$ for matrix2. Whenever a fiber is processed, a row from matrix1 is loaded and whenever a non-zero element is processed, a row from matrix2 is loaded. There are $SlcCnt \times ColCnt$ stores and $2 \times ColCnt \times (FbrCnt + NnzCnt)$ operations. For each fiber, *ColCnt* MACs and for each non-zero element, *ColCnt* more MACs.

Table 3 provides some insight on how the dataset affects the AI of the design. The AI is at its lowest when *NnzCnt* and *ColCnt* are at their minimum, i.e., one. Since there is only one non-zero element this causes *SlcCnt* and *FbrCnt* to also be one, as only slices and fibers with at least one non-zero element are stored. As for the maximum AI, *ColCnt* and $(FbrCnt + NnzCnt)$ must be as large as possible. On the other hand, *SlcCnt* must be as small as possible in order to decrease the number of loads, i.e., all non-zero

elements are in the same slice. In what concerns its peak performance, this design uses two DSPs for every column of the matrices, therefore one PE uses $2 \times ColCnt$ DSPs and performs $4 \times ColCnt$ operations per cycle.

5. Experimental Results

A direct performance comparison between the previously elaborated data-parallel approaches, for TTM and MTTKRP, also depends on factors other than the kernels' AI ranges, such as the processing capabilities of the device performing the computations (e.g., multi-core CPU or GPU), as well as on the characteristics of the sparse tensor dataset under evaluation. Therefore, we aim at describing the behaviour of the aforementioned kernels as well as uncovering their performance upper-bounds. For this purpose, we construct a set of synthetic sparse tensors in such a way that the worst-case and best-case performance can be attained. These synthetic best-case and worst-case sparse tensors are constructed based on the AIs, on Table 2, as well as on the architectures of the CPU and GPU devices.

To identify the performance upper bounds, it is necessary to create a tensor that enforces load-balancing as well as data locality. Hence, the prime candidate to achieve maximum performance of the TTM kernels is a semi-sparse tensor. This tensor is sparse in all its dimensions except for one, meaning there are several dense fibers sparsely scattered across the tensor. For the MTTKRP kernels, the tensor that maximizes the performance consists of several dense slices sparsely scattered across the tensor.

Following a similar reasoning as for the best-case, it is also possible to determine the performance lower bound on both CPU and GPU architectures. The strategy followed herein aims at analysing the worst-case scenario under the condition that the full utilization of processing resources is attained with a data distribution in the specifically created synthetic sparse tensor that hinders performance. The idea behind the worst-case scenarios is to prevent any data reuse, thus limiting the kernel's performance to the lowest bandwidth available on the architecture. To further accentuate the worst-case performance scenario, a lower AI is also desirable, hence a synthetic worst-case sparse tensor must have long fibers, each with a single non-zero element. It is worth noting that the non-zero elements are displaced across fibers in such a way that they do not allow for any reuse of the matrix elements. For MTTKRP, the worst-case tensor follows a similar pattern, i.e., large slices with a single fiber (conversely, a long fiber with a single non-zero element). In addition, both fibers and non-zero elements are displaced across the slices and fibers, respectively.

We also analyse our implementations with real-world execution scenarios. We performed tests on an Intel Core i9-11900KB multi-core CPU (16 threads at 3.30 GHz), with an integrated GPU, Intel 11th Gen UHD Graphics (32 Execution Units at 1.45 GHz). For the FPGA device, we used Intel Arria 10 GX 1150, with 427200 Adaptive Logic Modules (ALMs), 1708800 Registers and 1518 DSPs with a maximum frequency of 450 MHz. All experiments refer to the single-precision floating-point data and the performance numbers are averaged over five runs. Real-world sparse tensors from the FROSTT dataset [36] are shown in Table 4.

5.1. CPU Results

Figure 2 portrays the CARM characterisation of both our TTM kernels. The highlighted zone represents the AI range for the kernels and, from it, it is possible to observe that the kernels are memory bound for all memories except for L1 cache. It is important to notice that the performance for tensors nell-2 and vast-3D is within the range defined by the upper and lower bounds as expected. For MTTKRP, the performance attained for real-world tensors is also within the ranges defined by the best and worst-case scenarios, as can be observed in Figure 3. Tensor vast-3D provides better locality in the more frequent accesses to matrix2, hence providing a better utilisation of the cache hierarchy

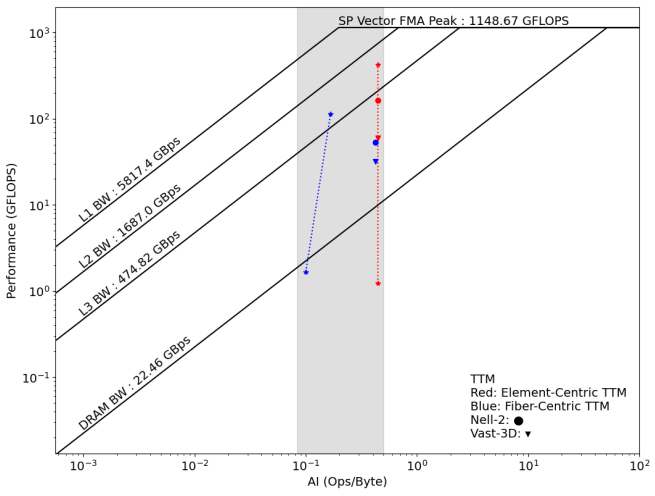


Figure 2: CARM for TTM on the CPU

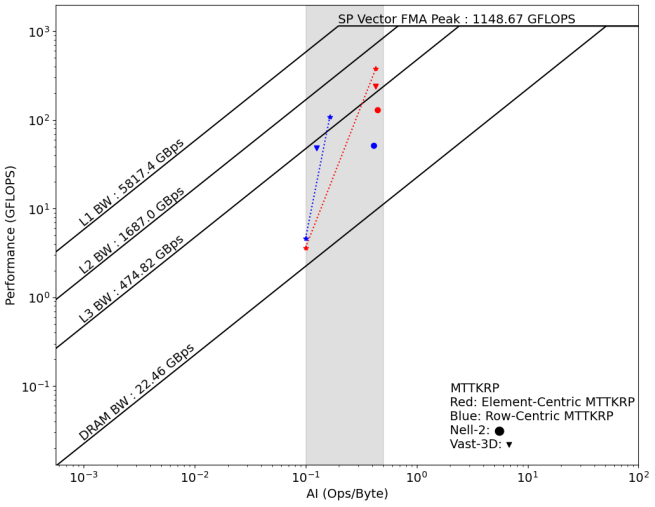


Figure 3: CARM for MTTKRP on the CPU

TABLE 4: Description of used sparse tensors

	SlcCnt	FbrCnt	NnzCnt	Mode 0	Mode 1	Mode 2
vast-3D	165 427	26 021 945		165 427	11 374	2
nell-2	12 092	337 365	76 879 419	12 092	9 184	28 818
nell-1	2 902 330	17 372 417	143 599 552	2 902 330	2 143 368	25 495 389

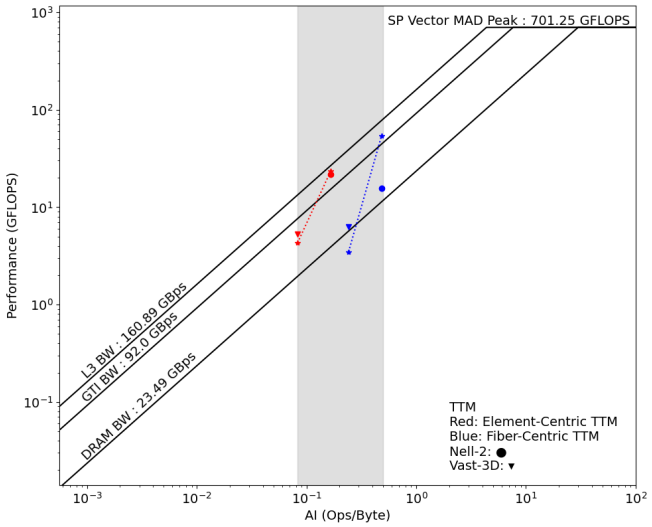


Figure 4: CARM for TTM on the GPU

in the CPU, which justifies the better performance when compared to the one attained in tensor nell-2. It is worth noting that on the CPU, compiler optimisations, namely the vectorisation of kernel loops, are important for both the AI and performance. For that reason, the Element-Centric approach is the one that performs better for both methods as the other approaches do not attain loop vectorisation.

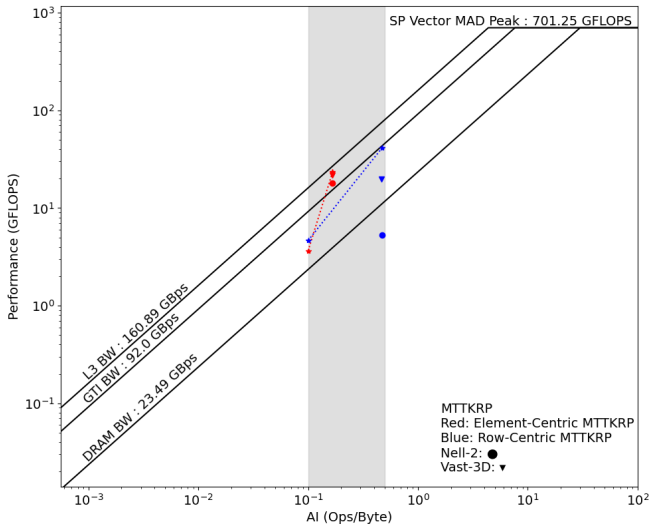


Figure 5: CARM for MTTKRP on the GPU

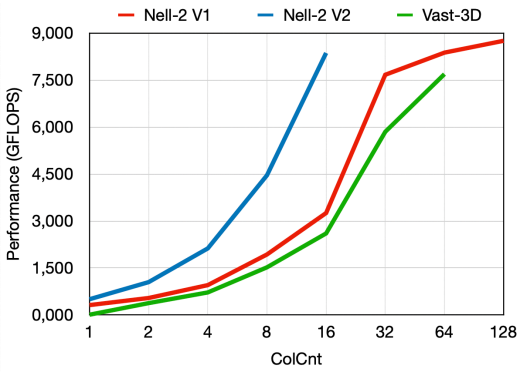


Figure 6: TTM performance on FPGA for varying number of columns

5.2. GPU Results

TTM and MTTKRP, on the GPU, are memory bound for all memories in the system, hence AI and memory access pattern are the determining factors for the performance on each dataset. From the CARM characterisation for TTM present on Figure 4, tensor nell-2 achieves an AI and performance close to the best-case scenario, only short due to the lack of data locality. On the other hand, tensor vast-3D achieves AI and performance close to the worst-case scenario, slightly better due to the matrix fitting in L3 cache. For the MTTKRP characterisation on the same device, Figure 5, the AI is close to the theoretical maximum for both datasets. Performance-wise, tensor vast-3D has displaced accesses to matrix1 and localised accesses to matrix2, in opposition to tensor nell-2, whose accesses are displaced for

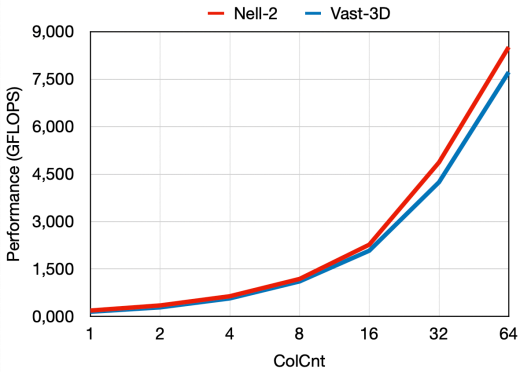


Figure 7: MTTKRP performance on FPGA for varying number of columns

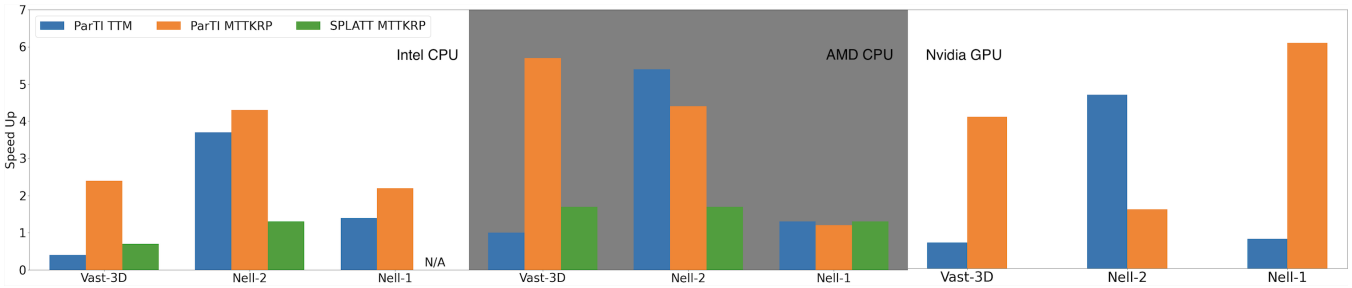


Figure 8: Speed Up over State of the Art

both matrices. Thus, the implementations achieving better performance for the tensor vast-3D.

5.3. FPGA Results

We deployed an tested our TTM and MTTKRP designs on a FPGA. Figures 6 and 7 portray the performance for these designs with a varying number of columns on the matrices. For TTM, we include the results for tensors vast-3D and nell-2. We also include, for tensor nell-2, the results for the same design with the matrix being loaded to on-chip memory before the computation. Performance increases exponentially for $ColCnt \leq 32$, since it is the maximum data-width the FPGA's load units can load in a single cycle. For the on-chip memory version, performance is always better than its counterpart, however the FPGA can only fit the design for $ColCnt \leq 16$. For MTTKRP, we also include results for tensors vast-3D and nell-2 with the performance displaying a similar, exponential, behaviour. The reason for the exponential growth of performance lies in the FPGA's programmable hardware. The number of DSPs in the design is adjusted to the number of columns in the matrices. There is, naturally, a limit to how many DSPs can be used.

In regards to the resource utilisation, it varies depending on the number of columns in the matrices. For $ColCnt = 16$, our first TTM design uses the following resources: 86033 ALMs, 129422 registers, 527 RAM blocks and 16 DSPs. While our design with on-chip matrix uses 85341 ALMs, 130635 registers, 1549 RAM blocks and 16 DSPs. When compared the major difference is, as expected, in the number of RAM blocks, since these are used to store the matrix. Our MTTKRP design uses 97834 ALMs, 143979 registers, 590 RAM blocks and 32 DSPs. Comparing to our first TTM design, it is important to notice the number of DSPs used doubles, as can be observed on Figure 1.

5.4. Comparison with Related Works

We focus herein in comparing our implementations to some of the state-of-the-art implementations, SPLATT [23] for MTTKRP on the CPU and ParTI [24] for both methods on CPU and GPU. From our implementations, we decided on the Element-Centric TTM and MTTKRP. These provide better performance across the tested datasets and devices. For modern CPU architectures, we resorted to Intel Core i9-11900KB and to AMD EPYC 7B13. For GPU architectures, we had to resort only to Nvidia A100 - 40GB, since the state-of-the-art implementations are in vendor-specific CUDA.

Figure 8 portrays the average speed up of our implementation over the state-of-the-art implementations for the datasets presented in Table 4. For each of the tensors, we measured execution times with different numbers of columns in the matrices. As can be observed, our implementation achieves better performance on both CPUs, except for TTM on tensor Vast-3D. This tensor has a single non-zero element per fiber, thus there is no advantage in using the CSF format over the COO format (used by ParTI).

Nvidia GPUs imposed further limitations when running our SYCL kernels. Besides the already present limit in the number of threads per work-group, a limit in the number of work-groups is also imposed. This forced our implementation to launch multiple kernels for a single tensor, hence hindering the performance. Still, it is possible to observe that for the tested datasets our implementation's performance is either comparable or better than the state of the art (up to 6×).

6. Conclusion

The main goal of this study was to delve into sparse tensor computations, namely *TTM* and *MTTKRP*, on heterogeneous systems. To achieve that, a detailed analysis of the algorithms' behaviour on the most common computational architectures was made. For the programmable devices, we exploited the available parallelism and data locality in memory accesses. To provide further insight on the algorithms, we built a set of synthetic tensors to validate our AI and performance predictions. This study, also featured designs for specialised architectures, which, by utilising their programmable hardware, are the most scalable implementation. Experimental results have shown the proposed sparse tensor methods can achieve, in multi-core CPU, GPU, heterogeneous and FPGA-based platforms, speedups of up to 6× for *TTM* and 7× for *MTTKRP*, when compared to the state-of-the-art approaches. Developing our implementations in SYCL assures portability, while creating a heterogeneous solution. While there is always potential for more optimisation, this study corresponds to a significant step towards sparse tensor computations on heterogeneous systems.

References

[1] Y. Wang, R. Chen, J. Ghosh, J. C. Denny, A. Kho, Y. Chen, B. A. Malin, and J. Sun, "Rubik: Knowledge guided tensor factorization and completion for health data analytics," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015, p. 1265–1274.

[2] J. C. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2014, p. 115–124.

[3] Y. Kwon, Y. Lee, and M. Rhu, "Tensor casting: Co-designing algorithm-architecture for personalized recommendation training," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 235–248.

[4] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights," *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[5] E. G. Hohenstein, R. M. Parrish, and T. J. Martínez, "Tensor hypercontraction density fitting. I. Quartic scaling second- and third-order Møller-Plesset perturbation theory," *jcp*, vol. 137, no. 4, pp. 044 103–044 103, jul 2012.

[6] F. Hummel, T. Tsatsoulis, and A. Grüneis, "Low rank factorization of the coulomb integrals for periodic coupled cluster theory," *The Journal of Chemical Physics*, vol. 146, no. 12, p. 124105, mar 2017.

[7] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Parcube: Sparse parallelizable candecomp-parafac tensor decomposition," *ACM Trans. Knowl. Discov. Data*, vol. 10, no. 1, jul 2015.

[8] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 598–611.

[9] M. Zhang, Z. Hu, and M. Li, "Duet: A compiler-runtime subgraph scheduling approach for tensor programs on a coupled cpu-gpu architecture," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 151–161.

[10] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowledge-Based Systems*, vol. 98, pp. 130–147, 2016.

[11] Y. Soh, P. Flick, X. Liu, S. Smith, F. Checconi, F. Petrini, and J. Choi, "High performance streaming tensor decomposition," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 683–692.

[12] S. Ma, X. Zhang, C. Jia, Z. Zhao, S. Wang, and S. Wang, "Image and video compression with neural networks: A review," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 6, pp. 1683–1698, 2020.

[13] N. C. Thompson, K. H. Greenewald, K. Lee, and G. F. Manso, "The computational limits of deep learning," *CoRR*, vol. abs/2007.05558, 2020.

[14] D. Blalock, J. J. G. Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" 2020.

[15] H. Farias, C. Nuñez, and M. Solar, "Tensorfit a tool to analyse spectral cubes in a tensor mode," *Astronomy and Computing*, vol. 25, pp. 195–202, 2018.

[16] M. Safayet Hossain, K. M. Azharul Hasan, and T. Tsuji, "Performance analysis of higher order tensor storages for highly sparse multidimensional data," in *2021 5th International Conference on Electrical Information and Communication Technology (EICT)*, 2021, pp. 1–6.

[17] Q. Sun, Y. Liu, H. Yang, M. Dun, Z. Luan, L. Gan, G. Yang, and D. Qian, "Input-aware sparse tensor storage format selection for optimizing mttkrp," *IEEE Transactions on Computers*, pp. 1–1, 2021.

[18] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, 2015.

[19] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, September 2009.

[20] V. T. Chakaravarthy, S. S. Pandian, S. Raje, and Y. Sabharwal, "On optimizing distributed non-negative tucker decomposition," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. Association for Computing Machinery, 2019, p. 238–249.

[21] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, P. Murali, S. S. Pandian, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for sparse tensors," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. Association for Computing Machinery, 2018, p. 374–384.

[22] L. Ma and E. Solomonik, "Efficient parallel CP decomposition with pairwise perturbation and multi-sweep dimension tree," *CoRR*, vol. abs/2010.12056, 2020.

[23] S. Smith and G. Karypis, "SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit," http://cs.umn.edu/ splatt/, 2016.

[24] J. Li, Y. Ma, and R. Vuduc, "ParTI! : A parallel tensor infrastructure for multicore cpus and gpus," Oct 2018, last updated: Jan 2020. [Online]. Available: http://parti-project.org

[25] H. Wang, W. Yang, R. Ouyang, R. Hu, K. Li, and K. Li, "A heterogeneous parallel computing approach optimizing spttm on cpu-gpu via gcn," *ACM Trans. Parallel Comput.*, feb 2023, just Accepted. [Online]. Available: https://doi.org/10.1145/3584373

[26] G. Xiao, C. Yin, Y. Chen, M. Duan, and K. Li, "Gsptc: High-performance sparse tensor contraction on cpu-gpu heterogeneous systems," in *2022 IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*, 2022, pp. 380–387.

[27] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu, "Performance gaps between openmp and opencl for multi-core cpus," in *2012 41st International Conference on Parallel Processing Workshops*, 2012, pp. 116–125.

[28] G. K. Reddy Kuncham, R. Vaidya, and M. Barve, "Performance study of gpu applications using sycl and cuda on tesla v100 gpu," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

[29] B. Madathil, S. V. M. Sagheer, A. Rahiman, A. J. Tom, B. P. S, J. Francis, and S. N. George, "Tensor low rank modeling and its applications in signal processing," 2019.

[30] A. Gonzalez, "Trends in processor architecture," 2018.

[31] M. Arora, "The architecture and evolution of cpu-gpu systems for general purpose computing," *By University of California, San Diago*, vol. 27, 2012.

[32] A. Boutros and V. Betz, "Fpga architecture: Principles and progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[33] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[34] R. Reyes, G. Brown, R. Burns, and M. Wong, "Sycl 2020: More than meets the eye," in *Proceedings of the International Workshop on OpenCL*, 2020.

[35] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[36] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/