A Distributed Framework for Low-Latency OpenVX over the RDMA NoC of a Clustered Manycore

Julien Hascoët[1,2], Benoît Dupont de Dinechin[1], Karol Desnos[2], Jean-François Nezan[2]
[1] Kalray, Montbonnot-Saint-Martin, France
[2] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, Rennes, France
{jhascoet, benoit.dinechin}@kalray.eu, {kdesnos, jnezan}@insa-rennes.fr

***Abstract*—OpenVX is a standard proposed by the Khronos group for cross-platform acceleration of computer vision and deep learning applications. OpenVX abstracts the target processor architecture complexity and automates the implementation of processing pipelines through high-level optimizations. While highly efficient OpenVX implementations exist for shared memory multi-core processors, targeting OpenVX to clustered manycore processors appears challenging. Indeed, such processors comprise multiple compute units or clusters, each fitted with an on-chip local memory shared by several cores.**

This paper describes an efficient implementation of OpenVX that targets clustered manycore processors. We propose a framework that includes computation graph analysis, kernel fusion techniques, RDMA-based tiling into local memories, optimization passes, and a distributed execution runtime. This framework is implemented and evaluated on the 2nd-generation Kalray MPPA® clustered manycore processor. Experimental results show that super-linear speed-ups are obtained for multi-cluster execution by leveraging the bandwidth of on-chip memories and the capabilities of asynchronous RDMA engines.

***Index Terms*—OpenVX, Low power, RDMA, Low latency, Tiling, Fusion, Prefetching, Memory wall, Fully automated**

I. Introduction

Server and desktop systems are built from multi-core processors that integrate up to a few tens of highly complex Central Processing Units (CPUs) cores. In order to improve energy efficiency while maintaining high computing performance, new processor architectures are designed with larger numbers of simpler cores [1], [2]. As the number of cores increases, however, it becomes beneficial to cluster these cores into compute units or clusters that become architecturally visible. Cores co-located into the same cluster may synchronize faster, may belong to the same coherency domain, and may share a local on-chip memory. GPGPUs are classic examples of manycore processors, whose compute units are called 'streaming multiprocessors', and where cores operate in SIMT (Single Instruction Multiple Threads) mode.

Our focus is on manycore processors built from fully programmable cores that operate in MIMD (Multiple Instruction Multiple Data) mode, and whose clusters include a RDMA engine able to move data asynchronously between the various on-chip and external memories. In particular, the Kalray MPPA2®-256 processor implements a clustered manycore architecture composed of 16 clusters of 16 cores each, interconnected with a Remote Direct Memory Access (RDMA)-enabled Network-on-Chip (NoC). Efficient programming of such manycore processors is challenging, as application software must distribute processing on the clusters and use the local memories as scratch-pad.

In the computer vision domain, open-source libraries like OpenCV are used for the rapid prototyping of applications on general purpose processors. With these libraries, application processing is expressed as a sequence of function calls, each implementing a black-box computation. This prevents classic compilation frameworks to perform global restructuring and high-level optimizations of the resulting applications. By contrast to OpenCV, the Khronos OpenVX standard [3] proposes a graph-based approach for the structured design of computer vision pipelines, where images flow as arcs between nodes, and nodes correspond to the processing kernels. Among other advantages, this approach enables implementations to expose and optimize the application at the graph level.

This paper describes a new OpenVX implementation for clustered manycore processors that performs high-level optimizations at runtime. These optimizations operate in a distributed framework for concurrent computations and asynchronous communications, with focus on low-latency execution of OpenVX applications. Optimizations include kernel fusion, asynchronous data prefetching, inter-cluster data transfers, multi-core scheduling, and memory allocation.

The organization is as follows: Section II presents background and related work regarding the OpenVX standard and other programming models. Section III describes our framework for clustered manycore architectures. Section IV presents experimental results on a Kalray MPPA2®-256 processor and discusses the strength and limitations of automatic optimizations. Finally, Section V concludes this paper.

II. Background and Related Work

A. Target Clustered Manycore Platform

The Kalray Massively Parallel Processor Array (MPPA)®2-256 "Bostan" processor integrates 256 VLIW processing cores and 32 VLIW management cores, each implementing the same ISA, on a single CMOS 28nm chip. It has been designed for high energy efficiency and time predictability for critical embedded applications [4]. Each of the 16 Compute Clusters (CCs) is composed of 16 processing cores sharing a multi-banked private local scratch-pad memory of 2MB. In addition, two Input/Output Subsystems (IOSs) are provided, each with two cache-coherent quad-cores implementing the same VLIW

ISA, sharing on-chip scratch-pad memory of 4MB, and connected to a DDR3 memory controller. In standalone operation of the processor, IOSs play the role of host multi-core CPUs for offloading computation onto the CC matrix. These 16 CCs and 2 IOSs are interconnected with a RDMA capable NoC. RDMA provides direct memory access from a local memory to another local memory, or between local memory and external memory, without involving any of the processing cores. Design and implementation of the RDMA over NoC library for the MPPA® processor are detailed in [5].

To the application programmer, the MPPA® processor CC matrix appears either as a single OpenCL [6] Compute Device or as a collection of individual multi-cores (one per CC). When used that way, the programmer has to instantiate a POSIX-like process on each CCs, where a lightweight executive implements a Pthread and OpenMP3 [7] multithreading runtime environment. Our OpenVX framework is built on the multiple multi-core view of the MPPA® processor.

B. The Khronos OpenVX Standard

The OpenVX standard [3] is a graph-based computing Application Programming Interface (API) proposed by the Khronos group for developing computer vision and deep learning applications on embedded platforms. OpenVX is not only designed as a host CPU acceleration model by a device like OpenCL but is also reminiscent of a dataflow Model of Computation (MoC). Dataflow MoCs are architecture-agnostic, highly valuable for exposing high-level optimization opportunities and enabling automatic deployment of applications on a wide variety of embedded platforms [8]. The OpenVX MoC appears a Single-Rate (SR) specialization of the Synchronous Dataflow (SDF) MoC [9], [8] where production and consumption rates of the graph nodes (actors) are equal. So a specific strength of OpenVX lies in exposing the graph structure of the entire processing pipeline, enabling implementations to perform high-level optimization and allowing vendors to get the most out of their machines.

An OpenVX application is given a *context* describing the accelerator device where the computation is to be offloaded. The OpenVX *Graph* is composed of *vertices* and *edges*. The *vertices* are OpenVX *Nodes*, which can be selected in a list of standard kernels [3]. The *edges* correspond to OpenVX buffers (*Images, LUTs, Arrays, Pyramids*, etc.) and link the *vertices* which produce and consume data. Two types of buffers exist: the user buffers, allocated and accessible from the memory space of the OpenVX host application; and the virtual buffers, that contain data exchanged between the *vertices* of the graph. Virtual buffers are not to be accessed by the host application and can be optimized away when fusing kernels.

C. OpenVX Implementations & Optimizations

An implementation of the OpenVX standard is proposed by virtually all IP vendors and chip producers that targets computer vision applications. OpenVX implementations also available from Graphical Processing Units (GPUs) and FPGA SoC vendors, where they share the offloading foundations of CUDA® or OpenCL. The Nvidia® VisionWorks framework presented in [10] implements OpenVX using CUDA for GPU offloading. The Advanced Micro Devices (AMD) open-source framework described in [11] uses either OpenCL for GPU offloading or the host CPU for computations. Both frameworks implement graph-based optimizations presented in [12] and target GPU-based accelerators or the host processor.

The ADRENALINE framework presented in [13] [14] features a series of optimization techniques including kernel fusion, overlap tiling by recomputing halo regions (ghost regions [15]), and double buffering for overlapping computation and communications. Seminal OpenVX optimizations techniques are described in [12], while the basics of efficient implementations on parallel machines are exposed in [16]: data prefetching, SIMD execution, and multiple levels of tiling. ADRENALINE provides a virtual prototyping platform currently implementing a single cluster and a host CPU. Their runtime is built on OpenCL 1.1 [17] with an extension to efficiently exploit the on-chip memory, avoiding round trips to the main memory whenever possible. By comparison, our work focuses on fully automating OpenVX graph optimizations and executing at low latency in a standalone mode (without external CPU). The standalone mode allows our framework to compile on-the-fly, with a call to *vxVerifyGraph* at runtime, the OpenVX graph onto the target processor if a configuration parameter of the OpenVX application changes dynamically. We instantiate a multi-core host CPU on one IOS, accelerated by up to 16 compute clusters, and we use asynchronous inter-cluster RDMA transfers to exchange halo regions.

While OpenCL can also be used for deploying kernels onto the compute matrix of the MPPA® processor, this standard does not support local memory sharing between kernels. Indeed, all `__global` data are committed back to the main memory, and `__local` data does not persist between kernels. This makes kernel fusion optimization impossible, with the result of global bandwidth becoming the main performance bottleneck. Vendor-specific extensions could be used to reuse memory between OpenCL kernels as in [14], but these are non-standard and not part of the Kalray OpenCL offer. Moreover, the Kalray OpenCL host runtime requires Linux which cannot be used for efficient, soft real-time systems because of process scheduling jitter and system call overhead.

III. Contributions

Our contribution is a framework for running OpenVX applications on stand-alone clustered manycore processors based on a distributed runtime execution environment. Starting from [18], which targets *Load/Store* CPU+GPU architectures with shared memory, we adapt and automate optimizations for both *Load/Store* (synchronous, intra-cluster) and RDMA (asynchronous, inter-cluster) types of memory accesses.

A. Compilation Workflow

The workflow specifies the automatic steps performed during the OpenVX graph on-the-fly compilation. The workflow is executed onto the embedded host; thus, graph recompilation

can be done at runtime if external parameters change. As shown in Figure 1, the workflow inputs the OpenVX application and produces computation commands for an accelerator.

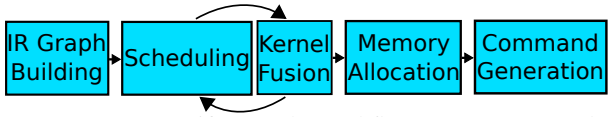

Fig. 1: OpenVX Verify Graph Workflow - *vxVerifyGraph* [3]

a) IR Graph Building provides the internal Intermediate Representation (IR), a Single-Rate (SR) Directed Acyclic Graph (DAG) on which the next passes of the compilation workflow operate. The graph builder takes user buffers which are OpenVX objects, searches for adjacent nodes using a Depth-First Search (DFS) and propagates application properties to buffers and nodes, such as image sizes and configuration parameters of OpenVX kernels. Several errors are detected and dealt with during the graph building process: unconnected buffers or nodes, cycles, multiple buffer writers, and the absence of input or output buffers for the OpenVX application. When errors are detected, the graph building results in failure giving the user the list of implicated nodes or buffers.

b) Scheduling is based on a topological sort of the SR-Directed Acyclic Graph (DAG) presented in [19]. It is performed to enforce the graph dependencies for kernel execution and has a complexity in $O(n)$.

c) Kernel Fusion is the key optimization that reduces the main memory bandwidth requirements, by running adjacent kernels on the same on-chip memory. Kernel fusion opportunities are identified by a simple constraint satisfaction algorithm that ensures memory allocation feasibility. The schedule is updated after each kernel fusion.

d) Memory Allocation pass is performed by a distributed memory allocator operating on the final schedule. As explained in III-E, virtual buffers are allocated to either the main memory or the array of symmetric scratch-pad memories.

e) Command Generation performs the computation of arguments for the RDMA-based tiling engine. The commands are saved in lookup tables. The runtime of the RDMA-based tiling engine running the compute clusters is presented in III-C. The basic tiling principle is to split a buffer into pieces such as slices or tiles and to distribute them onto computing resources. Once commands are generated, the *vxProcessGraph* consists in sending commands to the clusters explained in III-B. The commands are sent asynchronously but the execution is in schedule order across the matrix of compute clusters.

B. Graph Execution using an Efficient Offloading Engine

The deployment of computations from a host to one or several accelerators (compute clusters) is not a trivial task. The OpenVX application runs on the host multi-core CPU and invokes an acceleration API. The OpenVX *context* references the number of compute clusters in range $[0, 15]$ and the number of processors in range $[0, 15]$ inside each compute cluster of an MPPA® processor. Each OpenVX node is distributed on all available compute clusters (flat distribution). This is achieved by operating a low-level kernel offloading

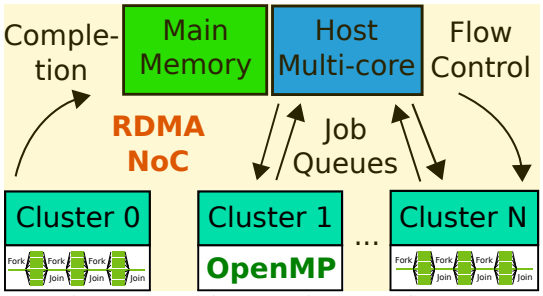

Fig. 2: Execution Runtime Architecture

engine in a lightweight multi-threaded runtime onto the host CPU. The offloading engine enables the deployment of self-synchronizing computations with efficient usage of the scratch-pad memories on the compute clusters. Figure 2 shows the offloading engine architecture where the OpenVX distributed framework is built. The parallelization relies on OpenMP3 # `pragma omp parallel` for work sharing between cores inside a compute cluster and uses the RDMA NoC API [5] to perform inter-cluster data transfers and main memory accesses.

All local memory accesses (intra-cluster) are done by *Load-Store*, and all remote memory accesses (inter-clusters) use RDMA for memory buffers and posted remote atomic operations for synchronizations. A local memory access is a low-latency memory transaction between a core and the internal cluster scratch-pad memory (10-cycles latency for *Load/Store*), whereas a remote memory access uses the RDMA protocol over the NoC to access another compute cluster memory or an off-chip memory (1200-cycles latency for *RDMA Put/Get*).

The design of our offloading engine is inspired by the GCC OpenACC [20] runtime back-end. Its implementation also heavily relies on the asynchronous one-sided communications and synchronization API over the NoC of the Kalray MPPA® processor, which provides high-throughput and low-latency RDMA, remote atomics and remote queue operations [5]. The distributed multi-cluster offloading engine provides the following set of features that are used by the generated compute commands of the OpenVX graph compilation workflow:

- Multi-cluster platform topology creation
- Load or unload code stream to the compute cluster(s)
- Scratch-pad buffer allocation associated to an identifier
- Execute kernel with arguments (name and arguments)
- Multi-cluster synchronization, synchronous or asynchronous collective
- Barrier of the computation pipeline, providing completion of outstanding kernels to the targeted clusters

All primitives are executed by the host multi-core CPU, asynchronously and atomically to avoid stalls and prevent data race respectively. However, initiated primitives are processed in the execution order on the compute cluster side. Thus, pipeline barriers are provided to ensure the completion of all outstanding primitives that were initiated to the targeted compute cluster(s). In this way, transactions are always pipelined for execution efficiency with regards to the host. Finally, the offloading engine provides implicit software flow-control mechanisms to prevent data corruption when the multi-cluster

system suffers from congestion. As a result, at 500 MHz, the measured Input/Output Operation per Second (IOPS) from the host point of view is 731.3 KIOPS, meaning an asynchronous request to a compute cluster takes 681 cycles on average.

C. Runtime Optimization RDMA-based Tiling & Fusion

The RDMA-based tiler operates at runtime (graph execution) inside each compute cluster concurrently, **distributing the execution of each OpenVX node across the entire matrix of compute clusters**. This technique is essential to achieve low-latency execution and contrasts with classic dataflow graph execution where actors map to clusters [2], [21]. Algorithm 1 receives commands through the job queues as seen in Figure 2 when the host application calls *vxProcessGraph*. Command arguments are the input and output images, tile geometries, halo geometries, the N-buffering configuration to absorb main memory latency, the start compute offset in main images for each compute cluster and the number of compute clusters that will execute the kernel concurrently.

First, the distributed tiler either retrieves input tiles from the main memory using N-buffering or sets local multidimensional input pointers to previous local output buffers of a previously executed kernel when it is fused with the current one. Second, the master thread on the compute cluster calls the compute kernel which performs intra-cluster parallelization with *pragma omp* directives. Third, the output is either committed back to the main memory for OpenVX user buffers or remains locally, if the next kernel is fused with the current one. When the next kernel is fused, depending on kernel fusion patterns, halo exchanges are initiated to adjacent compute clusters to satisfy inter-cluster data dependencies. Finally, memory consistency operations are initiated to memories that have outstanding writes before the multi-cluster synchronization.

Figure 3 shows an example of fusion in a 2D stencil computation using several clusters. A typical use case would be an edge detector followed by morphological operators. The black arrows are strided inter-cluster asynchronous RDMA transfers, also shown line 27 of Algorithm 1. Red arrows show global main memory spaced [5] transfers to the distributed array of scratch-pad memories. When executing a distributed fused kernel, each compute cluster is accommodated with tiles that are automatically reused from one kernel to the other. Therefore, the N-buffering N variable of Algorithm 1 is always set to 1 when fusing to maximize the on-chip memory usage, minimize data movement and save main memory bandwidth.

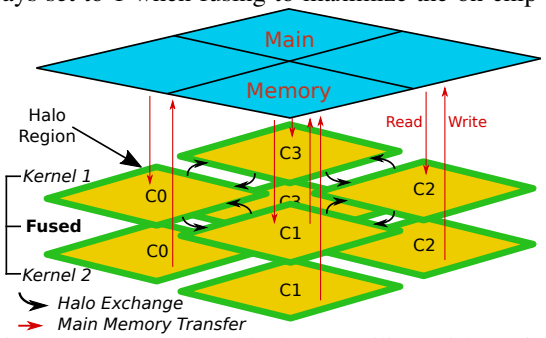

Fig. 3: Automated Multi-clusters Tiling with Fusion

Algorithm 1 An Automatic Distributed RDMA-based Overlap Tiler Concurrently Operating onto Multiple Compute Clusters.

1: **Input:** InImg, Width, Height, NbTotalTiles, N, TileWidth, TileHeight, HxIn/Out, HyIn/Out, NbTileStartOff, NbTiles
2: **Output:** OutImg
3: /* *Set multidimensional pointers in scratch-pad memory* */
4: Set InTiles[N][TileHeight+2*HyIn][TileWidth+2*HxIn]
5: Set OutTiles[N][TileHeight+2*HyOut][TileWidth+2*HxOut]
6: **for** i := 0 **to** N-1 **step** 1 /* *Warm up the pipeline* */ **do**
7: InTilesEvent[i] ← Asynchronous Get Stride-to-Dense from (InImg+NbTileStartOff+i) to InTiles[i]
8: **end for**
9: **for** i := N **to** NbTiles+N **step** 1 /**Pipeline Loop* */ **do**
10: ProcIdx := (i-N)%N /* *Compute Buffer Index* */
11: FetchIdx := i%N /* *Prefetch Buffer Index* */
12: /* *Wait for DMA Transactions Completions* */
13: Wait Get InTilesEvent[ProcIdx]
14: /* *Only one wait if in-place computation* */
15: Wait Put OutTilesEvent[ProcIdx]
16: /* *Compute Tile i-N in Parallel in the Node* */
17: OutTile[ProcIdx] := Kernel(InTiles[ProcIdx)])
18: **if** OutImg is local **then**
19: Async. Puts of halo regions to adjacent compute clusters for fused kernels dependencies
20: **else**
21: **if** i < NbTiles+N **then**
22: OutTiles[ProcIdx] ← Async. Put Dense-to-Stride to (OutImg+NbTileStartOff+i) from OutTiles[ProcIdx] /* *Write to Main Memory* */
23: **end if**
24: **end if**
25: /* *Prefetch Tile i from Main Memory* */
26: **if** i < NbTiles **then**
27: InTilesEvent[FetchIdx] ← Async. Get Stride-to-Dense from (InImg+NbTileStartOff+i) to InTiles[FetchIdx]
28: **end if**
29: **end for**
30: Async. Fence /* *Memory Consistency, Mandatory for Global Read-After-Write Dependencies* */
31: Synchronize NbNodes Clusters /* *Ordered with Fences* */

As explained in [22], reducing latency by software and hardware prefetching is a key to performance. Memory accesses are often the bottleneck in high-performance computing. Algorithm 1 has been designed to overcome the problem of hiding the external memory system access latency and exploiting inter-cluster RDMA data transfers to avoid the external memory bandwidth bottleneck. The RDMA-based tiler can be used straight out of the box by other architectures supporting (asynchronous) one-sided communications such as OpenCL `async_work_group_copy()`, MPI-3 one-sided

operations, or even the low-level onto the eDirect Memory Access (DMA) feature of the TI Keystone II.

D. Automatic Kernel Fusion Optimizations at Compile Time

The kernel fusion optimization consists of grouping two adjacent kernels together to avoid temporary buffers being copied to the external memory. Kernel fusion operates at multi-cluster level as each kernel is distributed on the whole compute cluster matrix to achieve low latency. It is inspired by the *Pairwise Grouping of Adjacent Nodes* algorithm, proposed in [9]. However, our kernel fusion is different, as each vertex of the SR-DAG is distributed on all available compute clusters, making data dependency between fused kernels a multi-dimensional problem as shown in Figure 3. Fusion decisions are based on the following constraints: the type of kernel pattern to fuse, the scratch-pad memory consumption and the type of input and/or output buffers have to be virtual. The fusion optimization pass, which has a complexity of $O(n)$, takes the main graph schedule as an input and produces a new schedule that represents the new fused kernels. This new schedule is then placed in the main graph schedule until all fusion opportunities are applied to the application graph. The scheduling policy consists of executing fused kernels in depth first. The supported patterns of kernel fusion are any combinations of point operator kernels using overlap tiling or not. The fusion optimization avoids recomputing halo regions and removes useless memory copies for the management of halo regions. However, it is involved regarding inter-cluster data transfers and the memory allocation of input and output tiles, as buffers need to be padded on the borders for halo exchange (See border of distributed tiles in Figure 3).

E. Distributed Static Memory Allocation at Compile Time

The distributed memory allocator manages the memory consumed by the virtual buffers of the OpenVX application. User buffers are already allocated at object creation. The distributed memory allocation operates after the scheduling and the kernel fusion pass. The allocator has two memory pools which are the array of symmetric scratch-pad memories and the main memory. The process of memory allocation is mainly influenced by the graph schedule through the lifetime of virtual objects, the kernel fusion decisions, the kernel dependency patterns, spills in the main memory for user buffers and also the N-buffering and tiling configurations parameters usually depending on image sizes. By default, the runtime automatically spills onto the main memory during the computation if there is not enough on-chip memory.

The RDMA-based tiler, described in Algorithm 1, is in charge of spilling and tiling images that do not fit into the available on-chip scratch-pad memories. Inside each compute cluster, a memory buffer of 1.4 MBytes is allocated at OpenVX context creation. This memory buffer size is configured in the OpenVX platform's specific files of the framework itself but is easily tunable to target any RDMA-enabled clustered manycores. This scratch-pad memory buffer accommodates temporary multidimensional sub-buffers that are allocated by a first-fit memory allocator giving buffer offsets in the scratch-pads. The first-fit algorithm takes buffers related to vertices in their schedule list order and recycles the memory once their live range has ended. On average, 4 sub-buffers are allocated in the scratch-pads before being recycled. Finally, the memory allocation is guaranteed to succeed as the kernel fusion optimization pass is aware of the available size remaining in the scratch-pads when fusing kernels. Indeed, when the kernel fusion takes too much memory, the fusing optimization pass chooses the RDMA-based tiler to spill on the main memory and to split the computation to make it fits automatically in the scratch-pad memories, thanks to Algorithm 1.

IV. Results & Discussions

Optimizations performed by the framework are fully automated without user intervention. This section shows the impact of automatic optimization passes regarding fusion and prefetching on the execution latency. The graph verification and scheduling were done offline during our benchmarking. The entire distributed framework (workflow, runtime, and kernels) has been implemented in standard C99 for efficient execution in embedded systems, and without any complex library dependency but the C library.

A. Performances Analysis

We use single-channel images (VX_DF_IMAGE_U8) for benchmarking with image sizes corresponding to VGA (480p) and full HD (1080p). Strong scaling is shown when varying the number of compute clusters, and the number of Processing Elements (PEs) is set to 16 within each compute cluster. The operating chip frequency is 500 MHz using a single DDR3 channel running at 1333 MHz. We use point operator kernels using tiling or overlap tiling techniques with either halo regions inter-cluster data transfers or spilling, depending onto the level of optimization.

1) Benefits of asynchronous RDMA prefetching: Figure 4 compares the execution latency of a single kernel using synchronous strided-to-dense RDMA main memory accesses compared to asynchronous accesses implementing N-buffering (see *bench*_N_BUFF results). We found that asynchronous RDMA prefetch is a must-have for performance, as long as the main memory is not the bottleneck. Quasi-linear speedups can be seen up to 8 clusters before becoming IO bound.

2) Automatic Kernel Fusion: As seen in Figure 5 for HD images with the *edge_detect_pipeline* (Median, Sharr and Magnitude pipeline), the fusion is automatically enabled when 10 computes clusters are used, eliminating temporary buffers to go back to the main memory, thus providing an extra speedup of 25 % in this use case compared to the spilling N-buffering version. With 10 compute clusters, the whole data set fits the sum of all available scratch-pad memories making the fusion optimization possible (Figure 3). Similar speedups are also noticed for other use cases for both image resolutions.

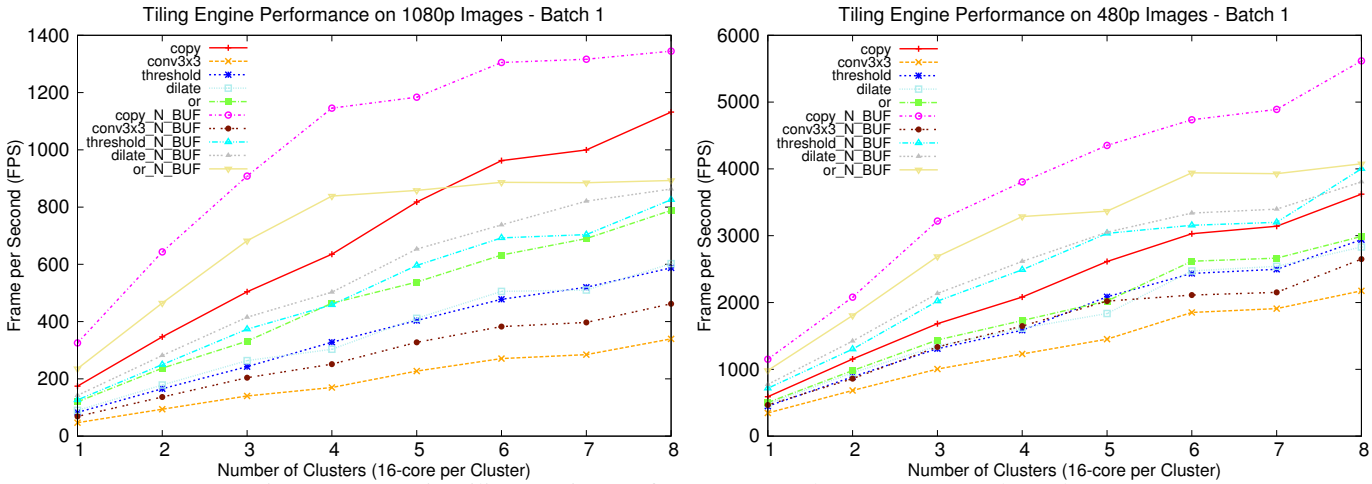

Fig. 4: Automatic Tiling Engine Performance. Batch 1: Latency = Throughput.
Simple Tiling vs Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch).

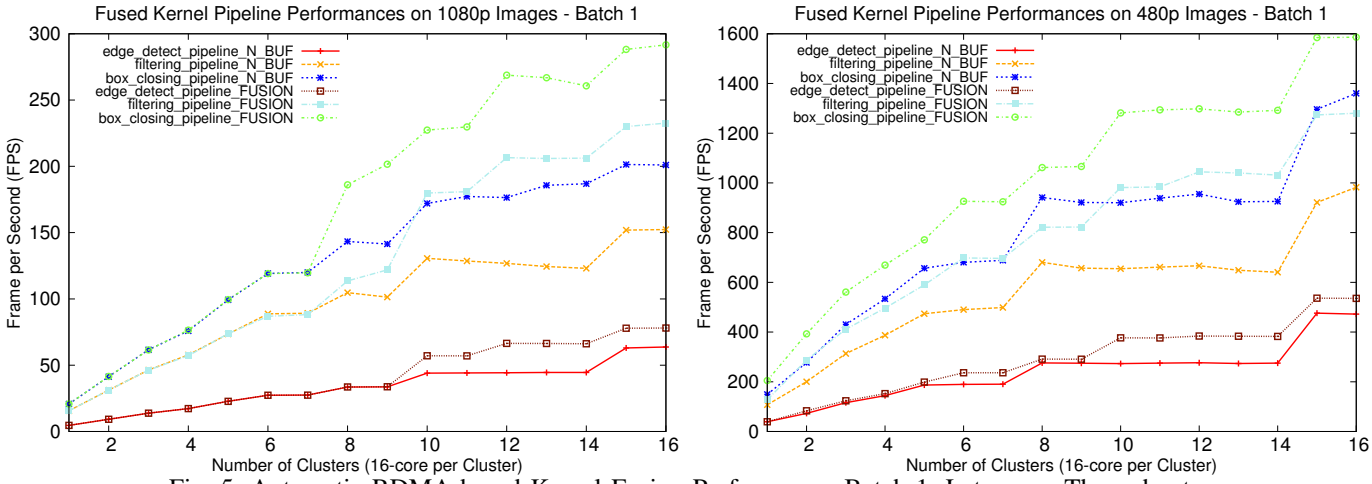

Fig. 5: Automatic RDMA-based Kernel Fusion Performance. Batch 1: Latency = Throughput.
Tiling with N-Buffering (N_BUF = N-Buffering = Prefetch) vs Kernel Fusing (FUSION).

3) Super-linear Speedup at Multi-Cluster Level: In the edge detection pipeline of Figure 5, super-linear speedups are observed. On full HD images (1080p), 1 compute cluster provides 4.62 frames per second (FPS) and the 16-cluster version with kernel fusions, asynchronous strided inter-cluster halo regions exchange provides 78.07 FPS, meaning a speedup of 16.9. Super-linear speedups are usually misunderstood as they contradict the classical theoretical speedup law's. On complex memory hierarchy processors, super-linear speedups can be achieved by optimizing multiple levels of memory access patterns in the memory hierarchy regarding the algorithm. Several parameters need to be taken into account, such as memory access locality (shared cache or scratch-pad usage), multiple levels of tiling geometries and asynchronous prefeching mechanisms. On the Kalray MPPA®2-256 processor, such speedup is obtained thanks to the exploitation of the on-chip scratch-pad memories, and the use of asynchronous (strided) inter-cluster data transfers which eliminate main memory access stalls. The memory bandwidth wall needs to be avoided to fully exploit the processing capabilities of low-power massively parallel architectures [16]. Other use cases *filtering_pipeline* (Sobel, Magnitude, Erode & Dilate) and *box_closing_pipeline* (Conv3x3, Erode & Threshold) show both speedups of 15.1 on full HD images.

V. Conclusion & Future Work

We describe a distributed framework for the low-latency implementation of the OpenVX computer vision standard on a clustered manycore processor operating in stand-alone mode. The main bottlenecks for the performance of applications are the limited external memory bandwidth and the long external memory access latencies. Our framework addresses both bottlenecks by exploiting the on-chip local memories as scratch-pad and by operating RDMA engines. Automated optimization techniques include kernel fusion, kernel execution tiling, and N-buffering of external memory transfers. Results measured on a silicon product show super-linear speedups at the multi-cluster level, demonstrating that the processor is well exploited. Future work will generalize the concept of RDMA-based time skewing for multidimensional problems as in [23] for cache-based machines, allowing kernel fusion optimization, independently of the available on-chip memories.

REFERENCES

[1] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit RISC system-on-chip," *CoRR*, vol. abs/1610.01832, 2016.

[2] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*. IEEE, 2013, pp. 1–6.

[3] K. V. W. Group *et al.*, "The openvx specification v1. 1," *Web: https://www. khronos. org/registry/OpenVX/specs/1.1/OpenVX_Specification_1_1. pdf*, 2017.

[4] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, 2015, pp. 220–229.

[5] J. Hascoët, B. D. de Dinechin, P. G. de Massas, and M. Q. Ho, "Asynchronous one-sided communications and synchronizations for a clustered manycore processor," in *Proceedings of the 15th IEEE/ACM Symposium on Embedded Systems for Real-Time Multimedia*. ACM, 2017, pp. 51–60.

[6] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, p. 66, 2010.

[7] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[8] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[9] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "Synthesis of embedded software from synchronous dataflow specifications," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 21, no. 2, pp. 151–166, 1999.

[10] F. Brill and E. Albuz, "Nvidia visionworks toolkit," in *GPU Technology Conference*, 2014.

[11] R. Giduthuri and K. Pulli, "Openvx: a framework for accelerating computer vision," in *SIGGRAPH ASIA 2016 Courses*. ACM, 2016, p. 14.

[12] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with openvx graphs," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 644–649.

[13] G. Tagliavini, G. Haugou, and L. Benini, "Optimizing memory bandwidth in openvx graph execution on embedded many-core accelerators," in *Design and Architectures for Signal and Image Processing (DASIP), 2014 Conference on*. IEEE, 2014, pp. 1–8.

[14] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Adrenaline: an openvx environment to optimize embedded vision applications on many-core accelerators," in *Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2015 IEEE 9th International Symposium on*. IEEE, 2015, pp. 289–296.

[15] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*. ACM, 2010, p. 4.

[16] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[17] K. O. W. Group *et al.*, "The opencl specification version 1.1," *http://www. khronos. org/registry/cl/specs/opencl-1.1. pdf*, 2011.

[18] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound gpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 191–202.

[19] D. J. King and J. Launchbury, "Structuring depth-first search algorithms in haskell," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 344–354.

[20] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012, pp. 859–870.

[21] L. Cudennec, P. Dubrulle, F. Galea, T. Goubier, and R. Sirdey, "Generating code and memory buffers to reorganize data on many-core architectures," in *Procedia Computer Science*, vol. 29, 2014, pp. 1123–1133.

[22] S. W. Williams, A. Waterman, and A. Patterson, "Roofline: an insightful visual performance model for floating-point program and multicore architecture," Technical report No. UCB/EECS-2008-134. https://www2. eecs. berkeley. edu/Pubs/TechRpts/2008/EECS-2008-134. pdf, Tech. Rep.

[23] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel, "Cache accurate time skewing in iterative stencil computations," in *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 2011, pp. 571–581.