embeddedworld2020
Exhibition&Conference
... it's a smarter world

Deep Learning Inference on the MPPA3 Manycore Processor

Benoît Dupont de Dinechin
Kalray S.A.
benoit.dinechin@kalray.eu

Julien Hascoët
Kalray S.A.
jhascoet@kalray.eu

Julien Le Maire
Kalray S.A.
jlemaire@kalray.eu

Nicolas Brunie
Kalray S.A.
nbrunie@kalray.eu

***Abstract*—We present the principles of deep learning inference on the MPPA3, Kalray's third-generation manycore processor. Unlike dedicated deep learning accelerators, the MPPA3 platform is also effective for image processing, signal processing, and numerical processing applications. This platform combines a CPU-based manycore architecture, processing elements composed of VLIW cores tightly coupled to tensor coprocessors, a standard-based programming environment capable of offloading C/C++/POSIX kernels, and a dedicated deep learning inference code generator. First deep learning inference results on the MPPA3 platform demonstrate higher performance and efficiency than a high-end embedded GPGPU platform.**

***Index Terms*—manycore processor, deep learning inference**

I. Introduction

In order to address the challenges of high-performance embedded computing with time-predictability, Kalray has been refining an homogeneous manycore architecture called MPPA (Massively Parallel Processor Array). The MPPA architecture applies the defining principles of manycore architectures: processing elements (SCs on a GPGPU) are assembled with a multi-banked local memory and a slice of the memory hierarchy into compute units (SMs on a GPGPU), which share an external memory and a global interconnect. The distinctive characteristic of the MPPA manycore architecture compared to GPGPU architectures is the integration of fully software-programmable VLIW cores for the processing elements (PEs), and the provision of a RDMA engine in each compute unit. The third-generation MPPA processor (Figure 1), manufactured in CMOS 16FFC technology, significantly improves over the previous generations in the areas of performance, programmability, functional safety, and cyber-security [1].

Although initially targeted to CPU-oriented (networking and storage infrastructure) and DSP-oriented (image and numerical processing) parallel workloads, the second-generation MPPA processor [2] has demonstrated significant potential for low-latency deep learning inference [3], thanks to the numerical performances of the VLIW cores, to the large on-chip local memories, and to the multicasting capabilities of the RDMA network-on-chip. Realizing this potential on the MPPA3 processor was achieved through architecture improvements, development of a unified programming environment for offloading C/C++/POSIX kernels, and re-engineering of the KaNN (Kalray Neural Network) inference code generator.

This paper discloses the features of the MPPA3 platform that enable high-performance and low-latency deep learning inference. Section II presents the MPPA3 architecture improvements for accelerated tensor processing. Section III introduces the standard-based programming environment for offloading C/C++/POSIX kernels, which applies to optimized libraries, to user application code, and to KaNN-generated code. Section IV describes the design and implementation of the KaNN deep learning inference code generator. First deep learning inference results are reported in Section V.

II. MPPA3 Tensor Processing

When considering deep learning acceleration, several architectural approaches appear effective. These include loosely coupled accelerators implementing a systolic datapath (Google TPU, NVidia NVDLA), coarse-grained reconfigurable arrays (Wave DPU, Cerebras WSE), or a bulk-synchronous parallel graph processor (GraphCore IPU). Other approaches tightly couple general-purpose processing units with vector or tensor processing units that share the instruction stream and the memory hierarchy. The latter approach has been instantiated in a peculiar way on the MPPA3 manycore processor.

On the MPPA3 processor [1], each VLIW core is paired with a tightly-coupled coprocessor for the mixed-precision matrix multiply-accumulate operations of deep learning operators (Figure 2). The coprocessor operates on a dedicated datapath that includes a 48×256-bit register file. Within the 6-issue VLIW core architecture, one issue lane is dedicated to the coprocessor arithmetic instructions, while the branch and control unit (BCU) may also execute data transfer operations between the coprocessor registers and the VLIW core general-purpose registers. Finally, the coprocessor leverages the 256-bit load-store unit (LSU) of the VLIW core to transfer data blocks from/to the local memory at the rate of 32 bytes per clock cycle. It then uses these 32-byte data blocks as either left or right operands of matrix multiply-accumulate operations.

The coprocessor datapath is designed assuming that the activations and weights respectively have a row-major and a column-major layout in memory, in order to avoid the complexities of Morton memory indexing [4]. Due to the mixed-precision arithmetic, matrix operands may take one, two or four consecutive registers, with element sizes of one, two, four, and eight bytes. In all cases, the coprocessor operations

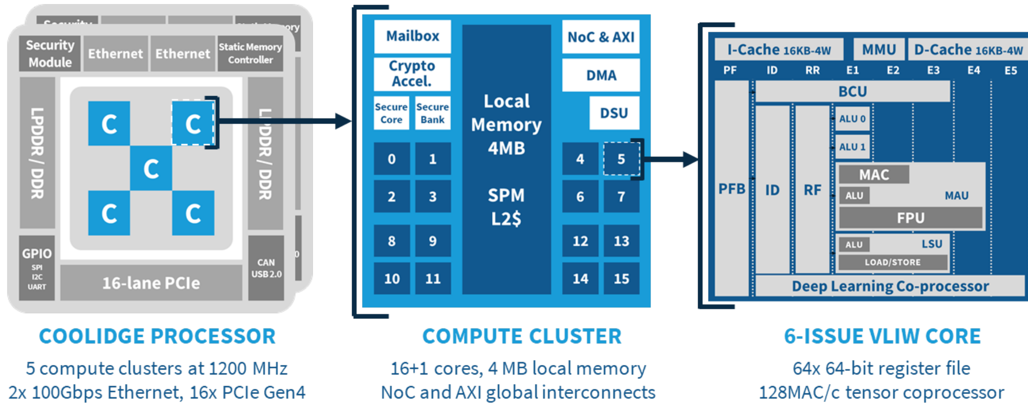

Fig. 1: Overview of the MPPA3 manycore processor.

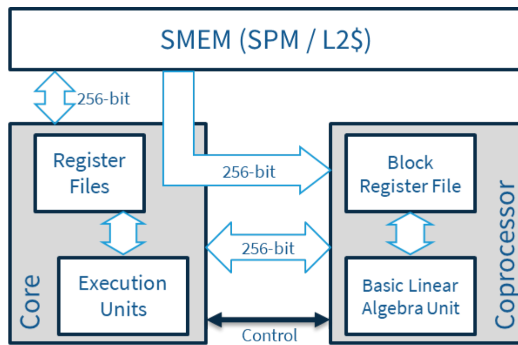

Fig. 2: Tensor coprocessor data path.

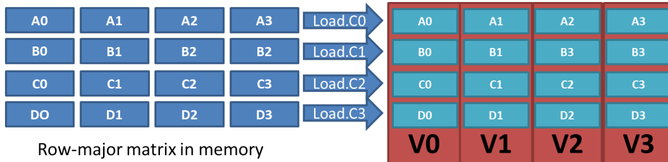

Fig. 3: Load-scatter to a quadruple register operand.

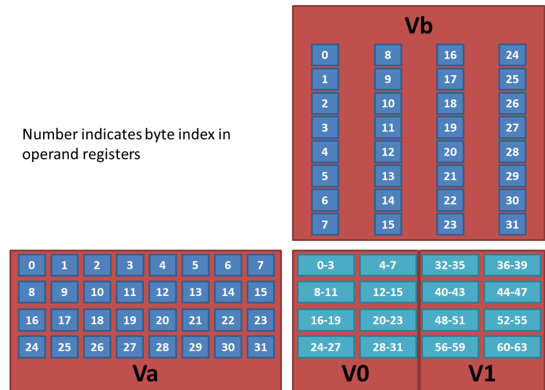

Fig. 4: INT8.32 matrix multiply-accumulate operation.

Fig. 5: INT16.64 matrix multiply-accumulate operation.

interpret matrix operands as having four rows and a variable number of columns, depending on the number of consecutive registers and the element size. In order to support this invariant, four 32-byte 'load-scatter' instructions to coprocessor registers are provided. A load-scatter instruction loads 32 consecutive bytes from memory, interprets these as four 64-bit (8 bytes) blocks, and writes each block into a specified quarter of each register that composes the destination operand (Figure 3). After executing the four load-scatter variants, a $4{\times}P$ submatrix of a matrix with row-major order in memory is loaded into a coprocessor register quadruple.

The coprocessor implements matrix multiply-accumulate operations on INT8.32, INT16.64 and FP16.32 representations[1]. The coprocessor is able to multiply-accumulate 4×8 by 8×4 8-bit matrices into a 4×4 32-bit matrix (128 MAC operations per clock cycle), held in two consecutive registers (Figure 4). The 8×4 8-bit matrix operand is actually a 4×8 operand that is transposed at the input port of the multiply-accumulate operation. The coprocessor may also perform multiply-accumulate operations of two 4×4 16-bit matrices into a 4×4 64-bit matrix (64 MAC operations per clock cycle), held in four consecutive registers (Figure 5). Finally, the coprocessor supports multiply-accumulate of two 4×4 FP16 matrices into a 4×4 FP32 matrix, but performed by four successive operations[2] (16 FMA operations per clock

[1]Numbers in each pair denote respectively the bit-width of the multiplicands and of the accumulator, while FP refers to the standard IEEE 754 binary floating-point representation

[2]Motivated by saving silicon area and not constrained by the architecture.

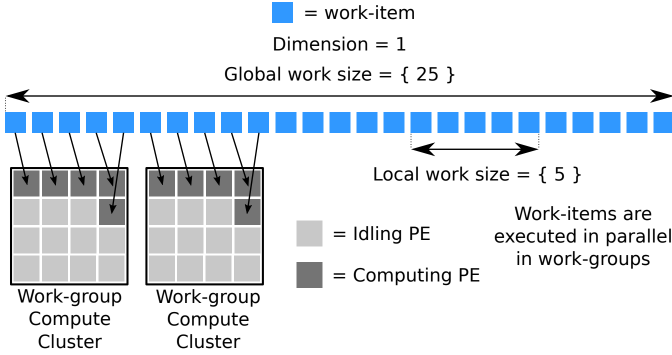

Fig. 6: OpenCL NDRange execution using SPMD Mode.

cycle). The FP16.32 matrix operations actually compute exact 4-deep dot-products with accumulation, by applying Kulisch's principles on a 80+ϵ accumulator [5].

III. Programming Environment

The programming environment used for deep learning inference on the MPPA3 processor is derived from the Portable Computing Language (PoCL) project[3], which proposes an open-source implementation of the OpenCL 1.2 standard[4] with support for some of the OpenCL 2.0 features. The OpenCL-C kernels are compiled with LLVM, which has been re-targeted for this purpose to the Kalray VLIW core. In OpenCL, a host application offloads computations to an abstract machine:

- An OpenCL device is an offloading target where computations are sent using a command queue.
- An OpenCL device has a global memory allocated and managed by the host application, and shared by the multiple compute units of the OpenCL device.
- An OpenCL compute unit comprises several processing elements (PEs) that share the compute unit local memory. Each OpenCL PE also has a private memory, and a shared access to the device global memory without cache coherence across compute units.

The OpenCL sub-devices are defined as non-intersecting sets of compute units inside a device, which have dedicated command queues while sharing the global memory.

In the port of the PoCL environment onto the MPPA3 processor, two offloading modes are provided:

LWI (Linearized Work-Items) All the work items of a work-group are executed within a loop on a single PE. This is the default execution mode of PoCL.

SPMD (Single Program Multiple Data)] The work-items of a work-group are executed concurrently on multiple PEs inside a compute cluster, where the `__local` OpenCL memory space is shared by the PEs and physically located in the local memory (Figure 6).

These mappings of the abstract OpenCL machine elements to the MPPA3 architecture components appear in Table I.

[3] `http://portablecl.org/`

[4] Passing the OpenCL 1.2 conformance with PoCL is work in progress.

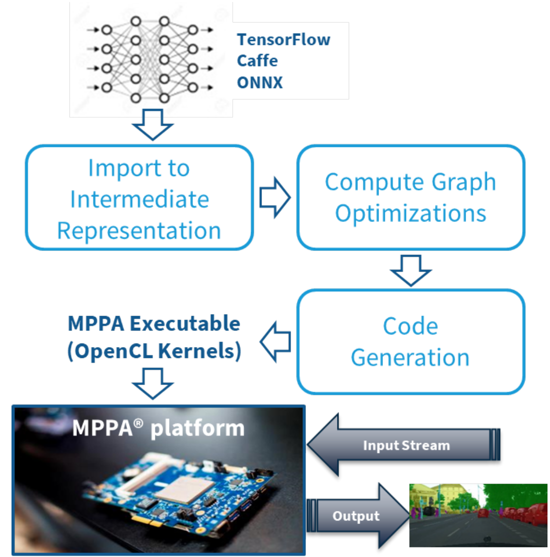

Fig. 7: KaNN inference code generator workflow.

On the MPPA3 processor, each deep learning inference application is dispatched to a partition composed of one or more compute clusters, which is exposed as an OpenCL sub-device executing in SPMD mode. Moreover, the code produced by the KaNN generator relies on the high-performance features implemented the GCC compiler for the Kalray VLIW core. Some of these features are not available in the LLVM compiler, in particular the support of the C named address spaces defined by ISO/IEC TR 18037:2008. The C named address spaces are used by the GCC compiler for the Kalray VLIW core to annotate objects and addresses that are accessed using *non-temporal* (L1D cache bypass) and/or *non-trapping* loads.

In order to call high-performance code compiled by GCC and MPPA communication libraries [6] from OpenCL-C kernels, the LLVM OpenCL-C compiler and PoCL have been extended to understand function declarations annotated with `__attribute__((mppa_native))`. Whenever such reference is seen in OpenCL-C source code, the PoCL linking stages assumes that the symbol is resolved, and the MPPA3 compute cluster run-time environment dynamically loads and links the native function before starting execution of the kernel. This native function extension also enables kernels to access other services such as a lightweight lock-free POSIX multi-threading environment, fast inter-PE hardware synchronisations, dynamic local memory allocation, and system call remoting to the host OS including FILE I/O.

IV. KaNN Code Generator

The KaNN (Kalray Neural Network) code generator is a deep learning inference compiler targeting the MPPA3 platform. It takes as input a trained neural network model described within a standard framework such as Caffe, TensorFlow or ONNX, and produces executable code for a set of compute clusters exposed as an OpenCL sub-device (Figure 7). Targeting OpenCL sub-devices allows several model inferences to execute concurrently on a single MPPA3 processor.

OpenCL	Device	Global Memory	Sub-device	Compute Unit
MPPA3 Component	MPPA Processor or MPPA Domain	External DDR Memory	Group of Compute Cluster(s)	Compute Cluster (SPMD) Processing Element (LWI)

TABLE I: Mapping of OpenCL machine elements to the MPPA architecture components.

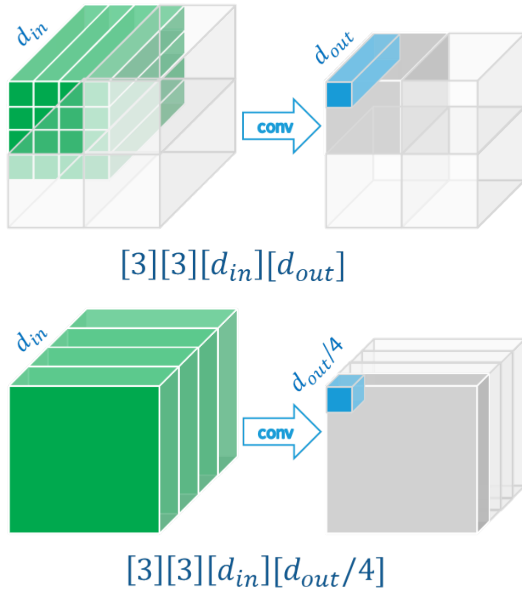

Fig. 8: Activation splitting across MPPA3 compute clusters.

The KaNN code generator optimizes for batch-1 inference, with the primary objective of reducing latency. At user option, FP32 operators in the original network can be converted to FP16 operators. Integer quantization such as the one used by TensorFlow Lite is also supported, however it must be expressed in the input model. Indeed, such models are assumed to be trained with fake quantization [7], which must match the actual quantization applied during inference.

Following import of the input model into an intermediate representation, optimizations are applied to the compute graph:

- elimination of channel concatenation and slicing copies;
- padding of input activations of convolutional layers;
- folding of batch normalizations, scalings, additions, into a single point-wise fused multiply-add operator;
- fusion of convolutions with ReLU activation functions;
- adaptation of arithmetic representations.

The KaNN code generation scheme is to perform inference in topological sort order of the (optimized) compute graph, parallelizing the execution of each operator over all the compute clusters of the target sub-device. When executing an operator, its input and output activations are distributed across the target local memories configured as SPM, while the network parameters are read from external DDR memory. Depending on the type of the operator (convolutional or fully connected), the spatial dimension sizes and the channel depth, input and output activations are distributed over the compute cluster local memories by splitting either along the spatial dimensions, or along the channel dimension (Figure 8).

- In case of spatial splitting of the output activations, each compute cluster only accesses an input activation tile and its shadow region, while all the operator parameters are required; these are read once from the DDR memory and multicasted to all the target compute clusters.
- In case of channel splitting of the output activations, the full input layer must be replicated in the local memory of each compute cluster, but only the corresponding slice of parameters is read from the DDR memory.

In all cases, activations are laid out in the local memory of the compute clusters along the channel dimension.

For any compute cluster in the sub-device, the code generation process defines and implements a local schedule for:

- local memory buffer allocations/de-allocations;
- DDR memory read/multicast of parameters;
- execution of operator operations;
- inter-cluster activation exchanges;
- and inter-cluster synchronizations.

This process is backed by the computation graph (Figure 9) augmented with parameter read tasks (yellow) and activation production tasks (blue).

The results of KaNN code generation is a collection of OpenCL binary kernels, where each kernel interprets the contents of a static data block composed of a sequence of records. Each record contains its length, a native compute function pointer, and a structure containing arguments for the compute function. For each record, the OpenCL kernel calls the native compute function with the pointer to the structure. The kernel ends after interpretation of the last record.

V. Experimental Results

The MPPA3 processor has been first demonstrated at the 2020 international Consumer Electronics Show (CES), running a Yolo v3 model inference for object detection and classification in the perception of a car autonomous driving system (Figure 10). The standard Yolo v3 configuration is used, with Darknet-53 as backbone and input image size of 416×416 [8]. Inference is performed using FP16.32 arithmetic, with KaNN targeting the 5 clusters of the MPPA3 processor, since an external host CPU is used. When operating at 1.2 GHz, the performance is close to 20 FPS (Frames Per Second).

It is interesting to compare this figure to the performances of GPGPUs. Yolo authors report[5] that the NVIDIA Pascal Titan X processes 416×416 images at 35 FPS, for a total of $65.86\ 10^9$ floating-point operations. A NVIDIA blog[6] mentions that the Jetson AGX Xavier processor runs Yolo v3 416×416 model inference with FP16 arithmetic at 18 FPS.

[5] https://pjreddie.com/darknet/yolo/

[6] https://devtalk.nvidia.com/default/topic/1058408/deepstream-sdk/yolov3-fps-on-xavier/