KALRAY

Parallel Processing with the MPPA Manycore Processor

Kalray MPPA®

Massively Parallel Processor Array

Benoît Dupont de Dinechin, CTO

14 Novembre 2018

www.kalrayinc.com

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

KALRAY IN A NUTSHELL

We design processors at the heart of new intelligent systems

4
offices
Grenoble, Sophia (France),
Silicon Valley (Los Altos, USA),
Yokohama (Japan)

~80 people
~70 engineers

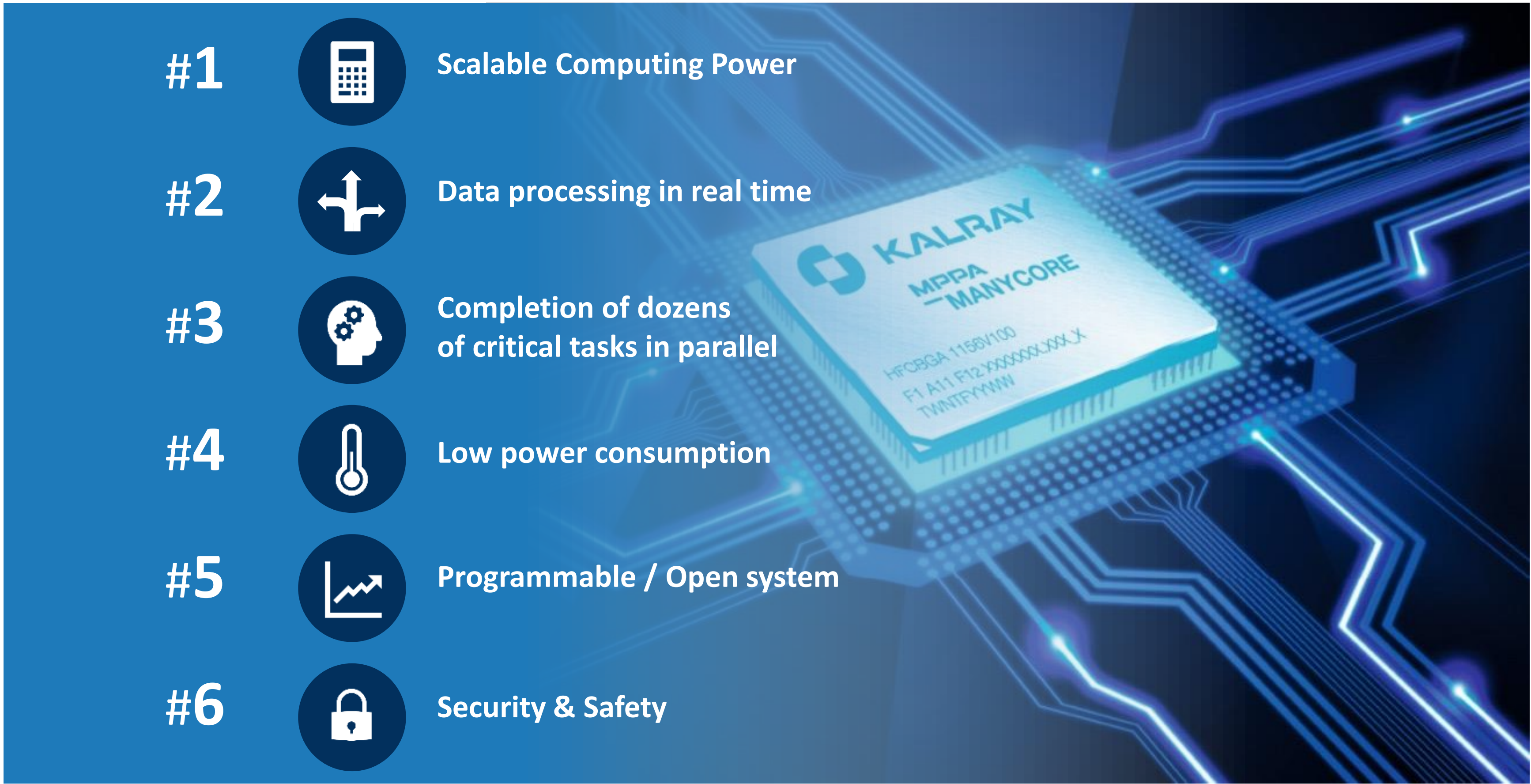
A unique technology, result of **10** years of development

Financial and industrial shareholders

cea investissement
INOCAPGestion
MBDA
RENAULT NISSAN MITSUBISHI
ACE
Pengpai
SAFRAN
bpifrance

ALKAL
EURONEXT
GROWTH

KALRAY: PIONEER OF MANYCORE PROCESSORS



#1 Scalable Computing Power

#2 Data processing in real time

#3 Completion of dozens of critical tasks in parallel

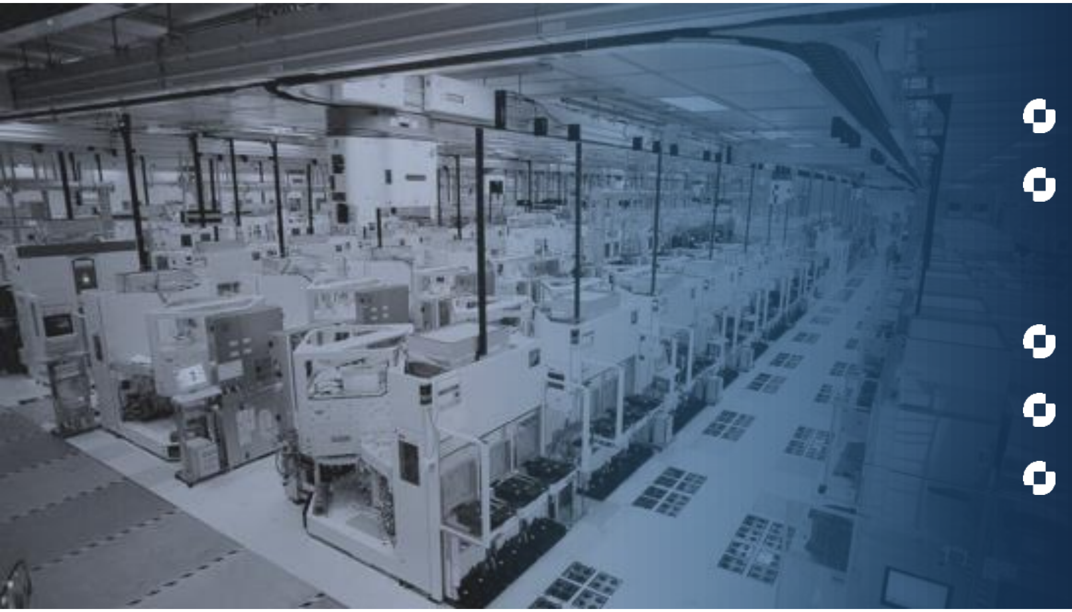
#4 Low power consumption

#5 Programmable / Open system

#6 Security & Safety

OUTSOURCED PRODUCTION
(A FABLESS BUSINESS MODEL)

PARTNERSHIP WITH THE WORLD LEADER IN PROCESSOR MANUFACTURING



- **Sub-contracted production**
- **Signed framework agreement with GUC, subsidiary of TSMC**
  (world top-3 in semiconductor manufacturing)
- **Limited investment**
- **No expansion costs**
- **Production on the basis of purchase orders**

INTELLIGENT DATA CENTER : KEY COMPETITIVE ADVANTAGES

- First “NVMe-oF all-in-one” certified solution *
- 8x more powerful than the latest products announced by our competitors**
- Power consumption below 20W***

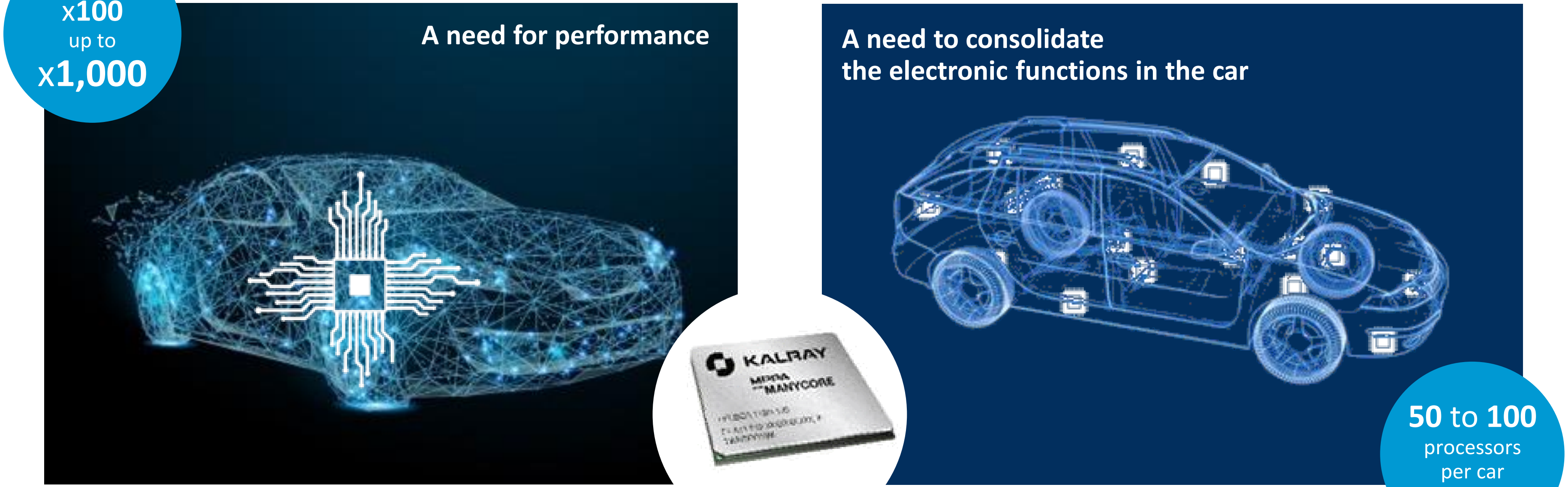
** Kalray KTC80 has been certified in April 2018 by the independent certification Inter Operability Laboratory ((University of New Hampshire). No competitors’ products has been certified so far (www.iol.unh.edu/registry/nvmeof)*
*** Kalray KTC80 : 288 cores @ 550MHz = 158GHz / Mellanox Bluefield : 16 cores @ 1.2GHz = 19.2GHz / Broadcom Stingray : 8 cores @ 2GHz = 16 GHz*
**** Kalray measurement of KTC80*

KALRAY: THE SOLUTION THAT BRINGS INTELLIGENCE “ON THE FLY” TO THE WORLD OF DATA CENTERS

OUR MPPA IS A UNIQUE SOLUTION TO ADDRESS TWO MAIN CHALLENGES FACED BY OEMs

PERFORMANCE

AGGREGATION



Perf. increase
x100
up to
x1,000

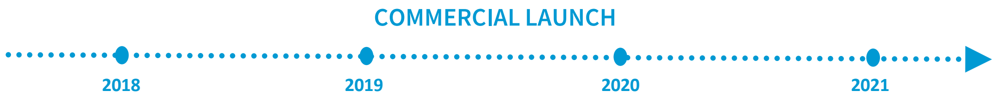
A need for performance

**A need to consolidate
the electronic functions in the car**

50 to 100
processors
per car
today

KALRAY MPPA
TECHNOLOGY

MPPA® PRODUCT FAMILY AND ROADMAP



	BOSTAN	COOLIDGE -1	COOLIDGE -2	Dx
PROCESS	28 nm	16 nm	16 nm	12 nm or 7nm
PERFORMANCE	1 DL TOPS 700 MFLOPS SP	24 DL TOPS 1 TFLOPS SP 3 TFLOPS HP	48 DL TOPS / 96 TDL OPS	100 TOPS / 200 TOPS
USE	Boards SC (40G) Prototypes	Boards and storage chip controllers (100G) Accelerator intelligent car	Qualification Car Market DC - NFV	DC
CONSUMPTION (WATTS)	8W – 25W	5W – 15W	5W – 20W	2W – 10W
		UNDER DEVELOPMENT		UNDER DEFINITION

MANYCORE TECHNOLOGY THAT ENABLES PROCESSOR OPTIMIZATION BASED ON EVOLVING MARKET REQUIREMENTS

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

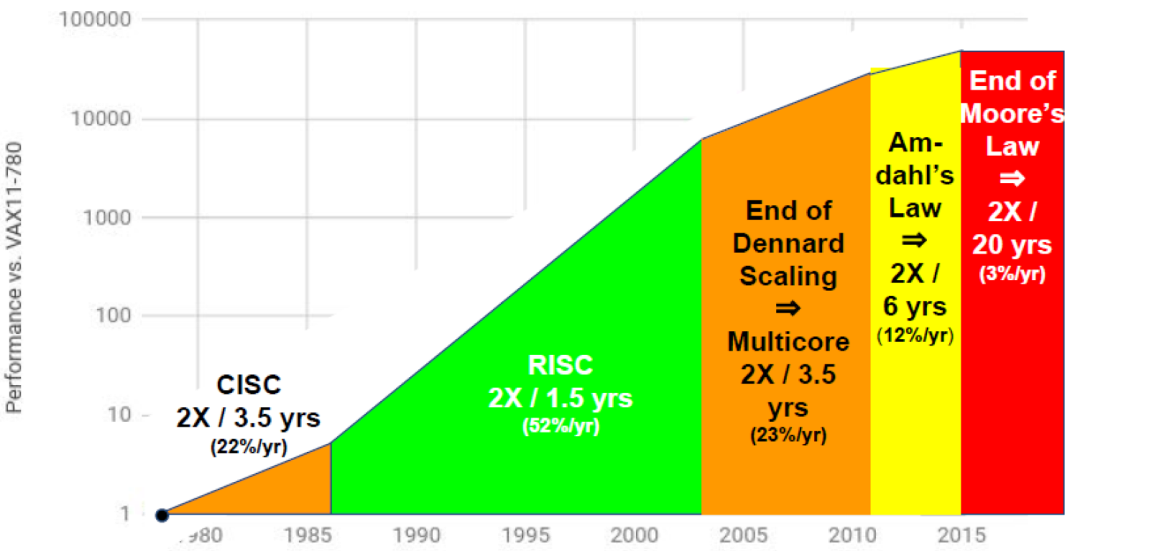
Model-Based Programming

Deep Learning Inference

Conclusions

Motivation for Multicore and Manycore Processors

Past contributions to CPU performances: clock speed increase, instruction-level parallelism, thread-level parallelism



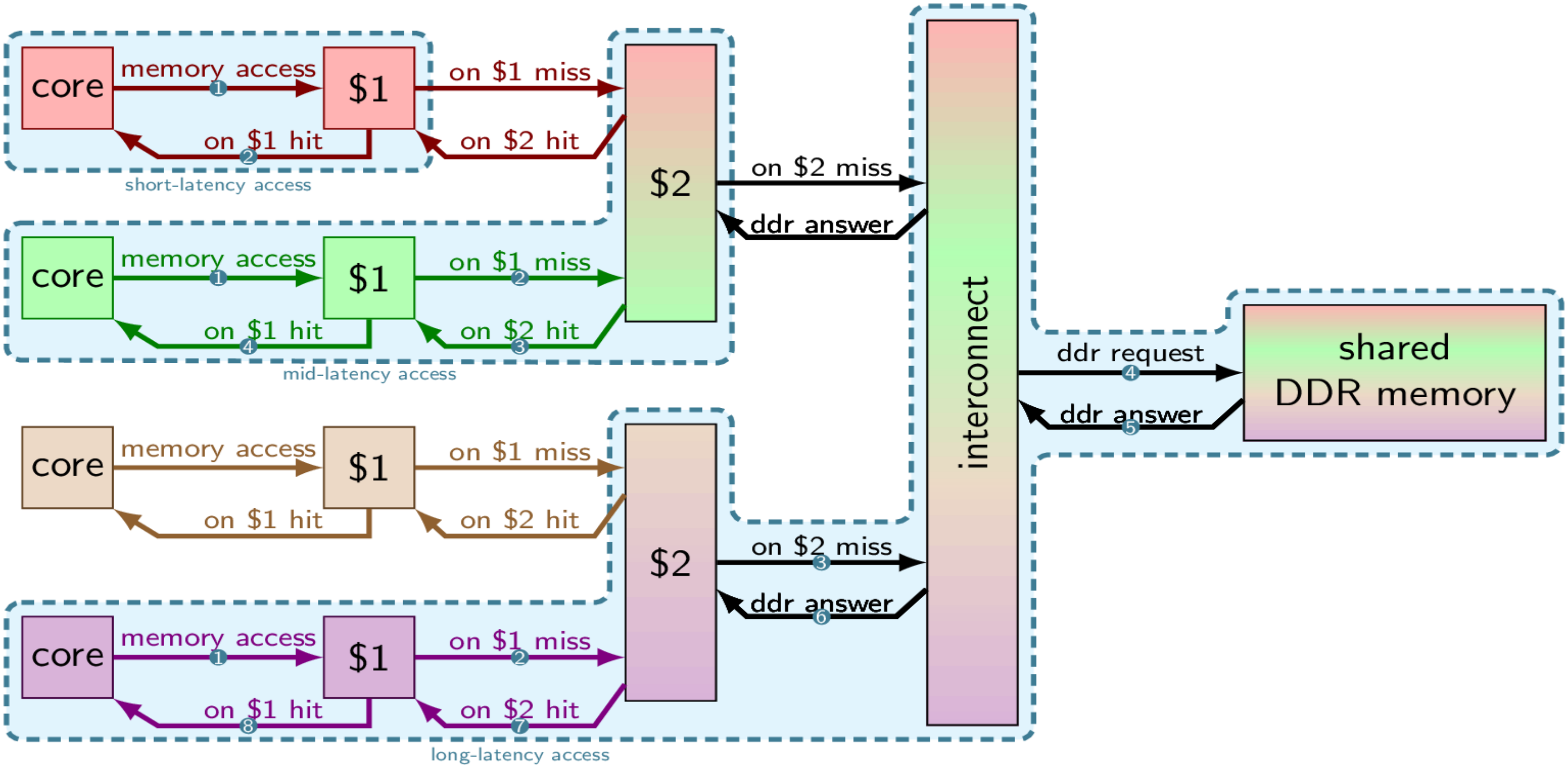
Based on SPECintCPU. Source: John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, 6/e. 2018

Intuitively, these are Manycore Processors

Processor	Cores	Year	Applications
Tilera/Mellanox TILE-Gx72	72x 64-bit VLIW cores	2014	Networking, Storage
Parallela Epiphany-V	1024x 64-bit RISC cores	2016	Embedded HPC
Intel Xeon Phi Knights Landing	72x Atom cores with four threads per core	2016	Supercomputing
Sunway SW26010 (TaihuLight)	260x 64-bit RISC cores	2016	Supercomputing
Kalray MPPA3-80 Coolidge	85x 64-bit VLIW cores	2018	Embedded HPC, Networking, Storage
REX Computing NEO	256x 64-bit VLIW cores	2018	Embedded HPC, Supercomputing
NVIDIA Xavier	512x 64-bit CUDA cores	2018	Embedded HPC

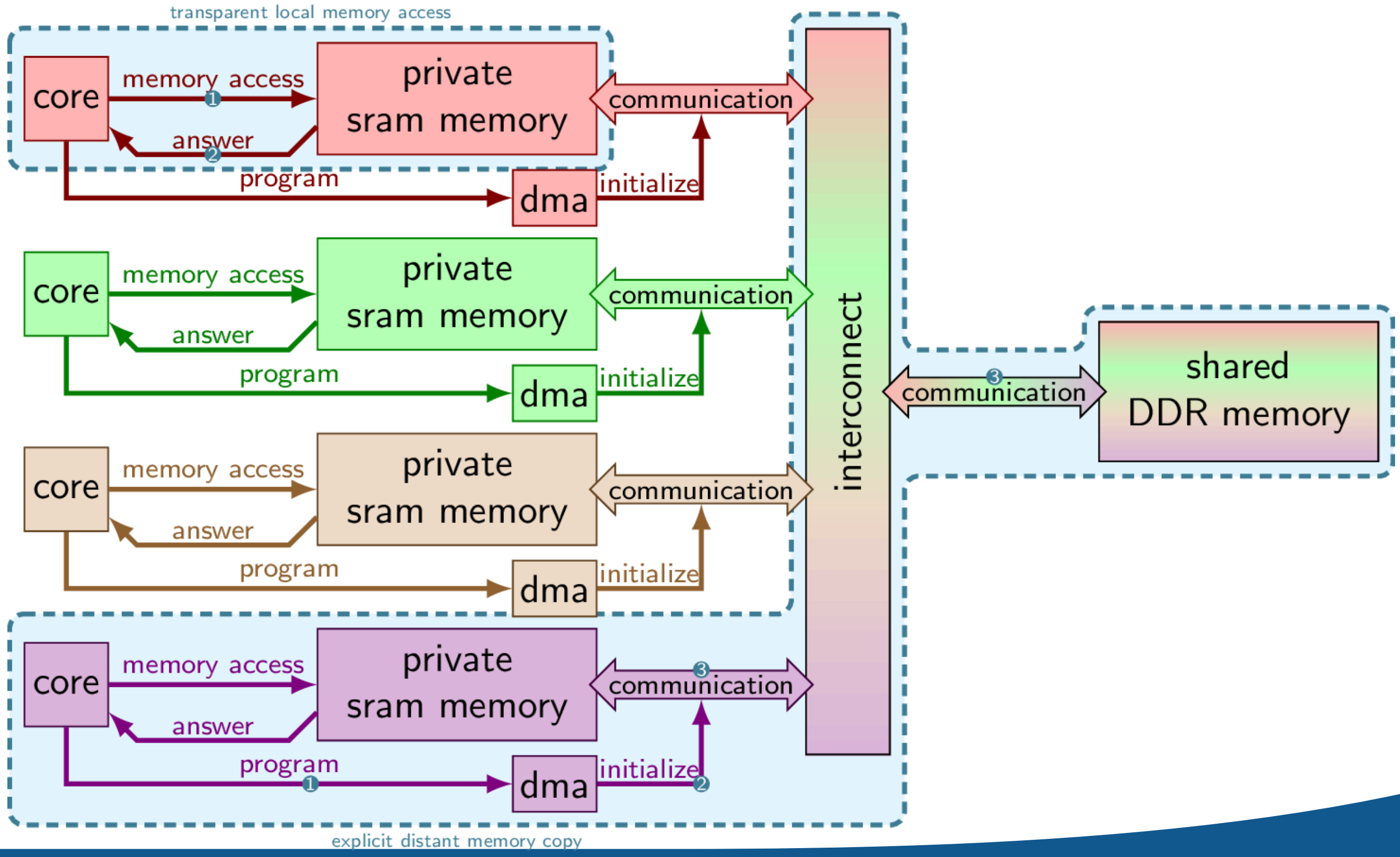
Classic Multicore Memory Hierarchy

Challenge: managing interference between cores



Embedded Multicore Memory Hierarchy

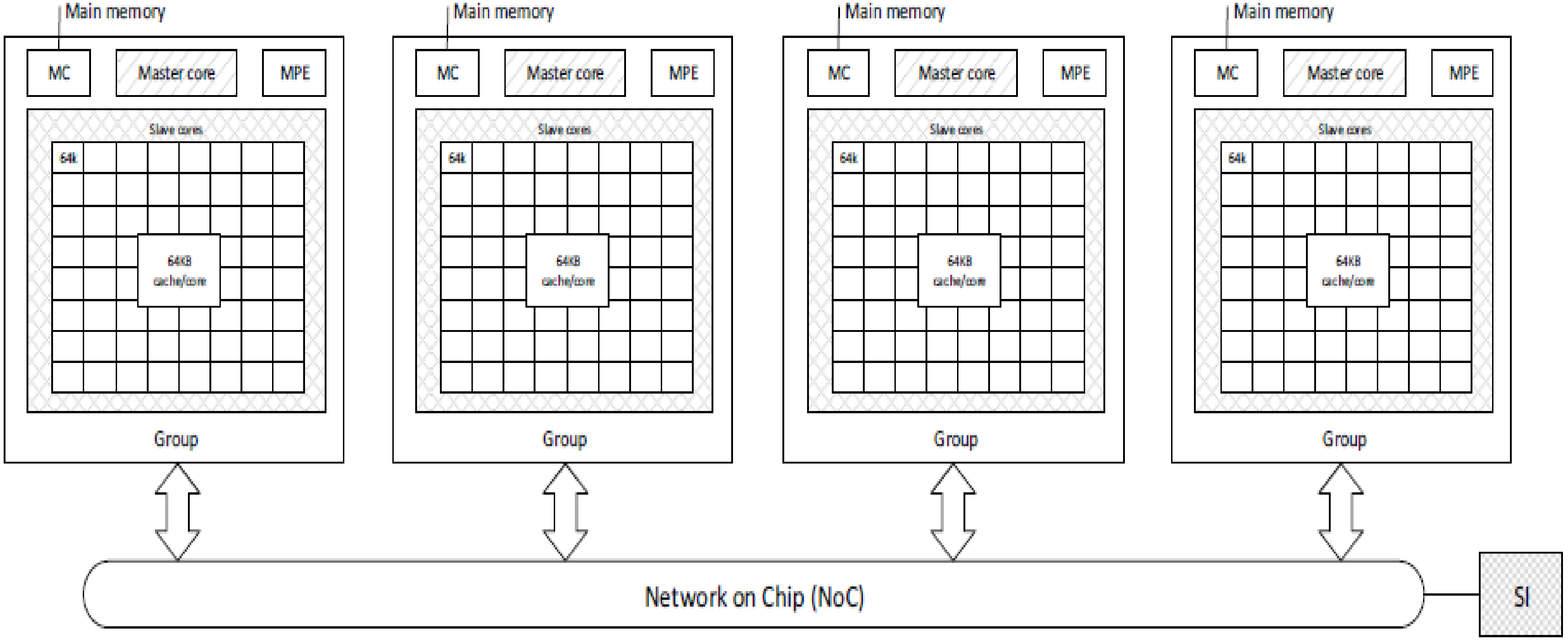
Challenge: programmability of DMA and private memories



A Qualitative Definition of Manycore Processors

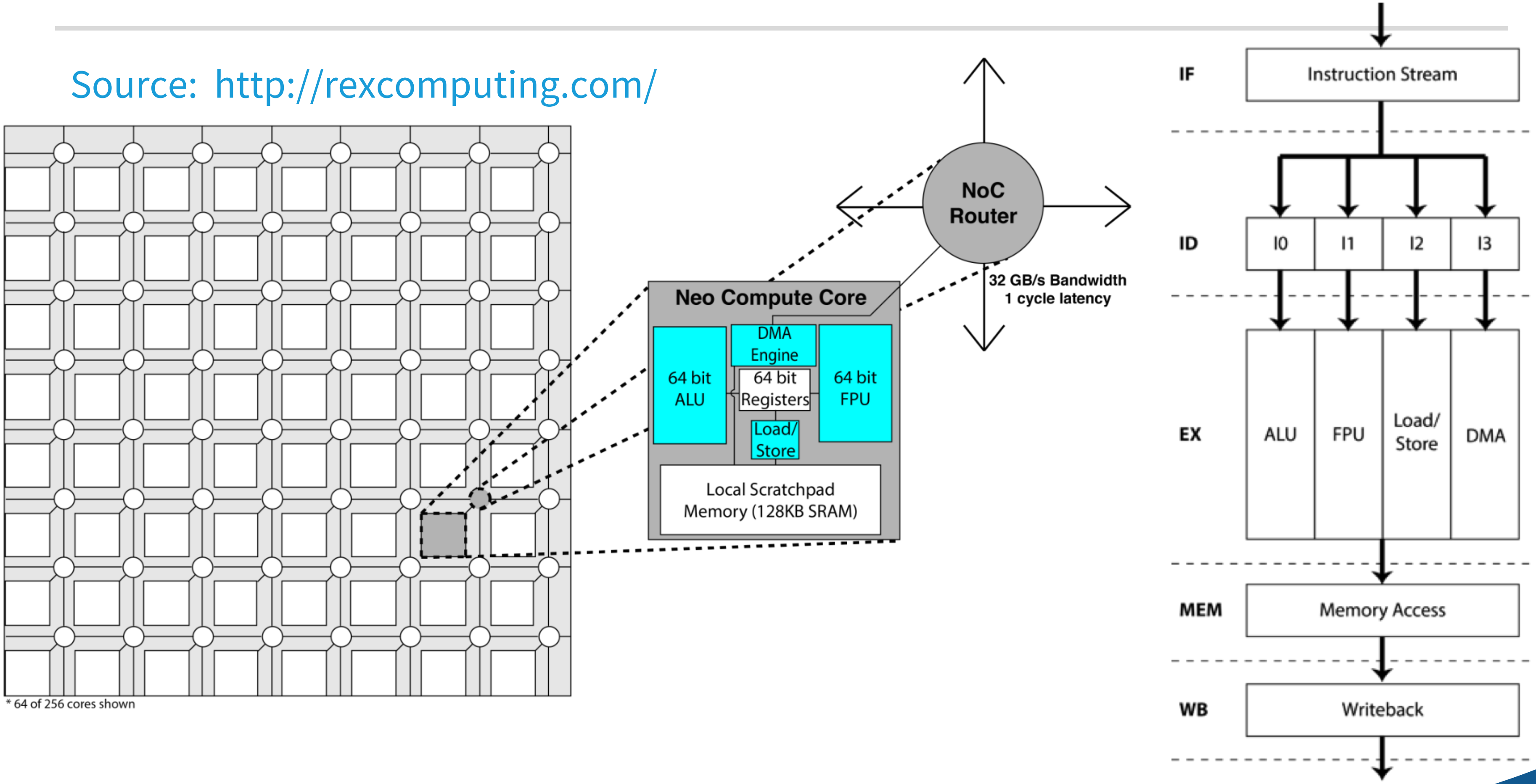
Memory locality and core clustering are architecturally visible

- Scratch-pad memory (SPM), software-managed caches, local memory, ‘shared memory’ (GPGPUs)
- ‘compute unit’ associates processing cores and data transfer engines operating on a local memory
- Sunway SW26010 processor with 64KB SPM per CPE core (source U. of Tennessee / Jack Dongarra):



REX Computing NEO Architecture (Defunct)

Source: http://rexcomputing.com/



* 64 of 256 cores shown

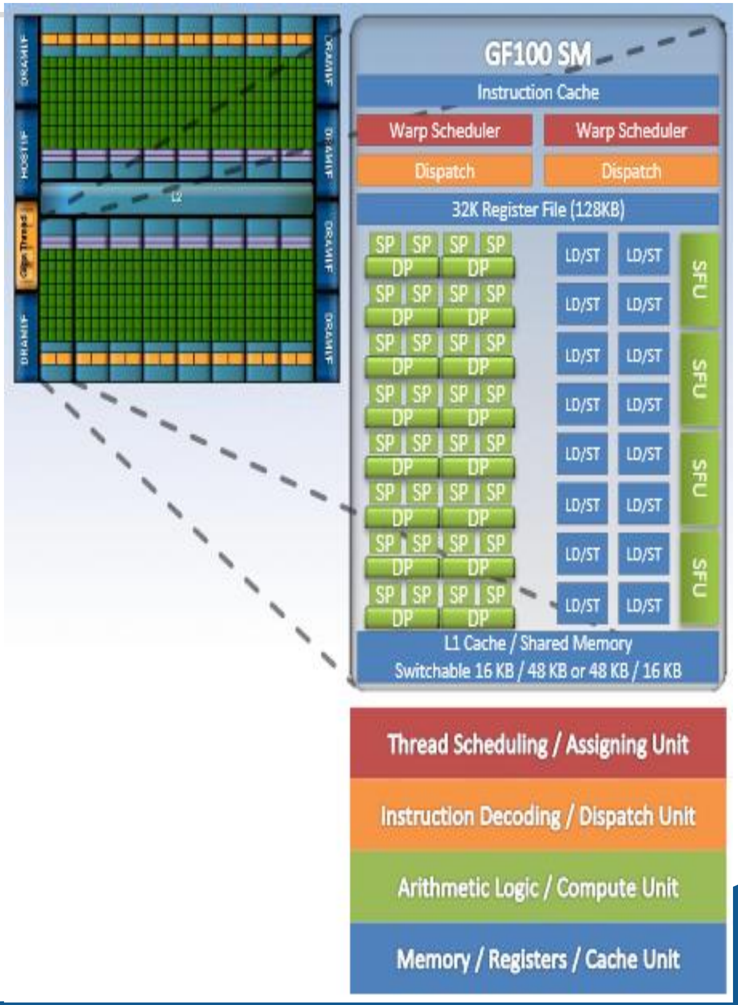
GPGPUs as Manycore Processors

Classic GPGPU architecture: NVIDIA Fermi

- GPGPU ‘compute units’ called Streaming Multiprocessors (SM)
- Each SM comprises 32 ‘streaming cores’ or ‘CUDA cores’ that share a local memory, caches and a global memory hierarchy
- Threads are scheduled and executed atomically by ‘warps’, where they execute the same instruction or are inactive
- Hardware multithreading enables warp execution switching on each cycle, helping cover memory access latencies

GPGPU programming models (CUDA, OpenCL)

- Each SM executes ‘thread blocks’, whose threads may share data in the local memory and access a common memory hierarchy
- Synchronization inside a thread block by barriers, local memory accesses, atomic operations, or shuffle operations (NVIDIA)
- Synchronization between thread blocks through host program or global memory atomic operations in kernels



DRAM I/F
HOST I/F
GigaThread
L2
GF100 SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch
Dispatch
32K Register File (128KB)
SP
DP
LD/ST
SFU
L1 Cache / Shared Memory
Switchable 16 KB / 48 KB or 48 KB / 16 KB
Thread Scheduling / Assigning Unit
Instruction Decoding / Dispatch Unit
Arithmetic Logic / Compute Unit
Memory / Registers / Cache Unit

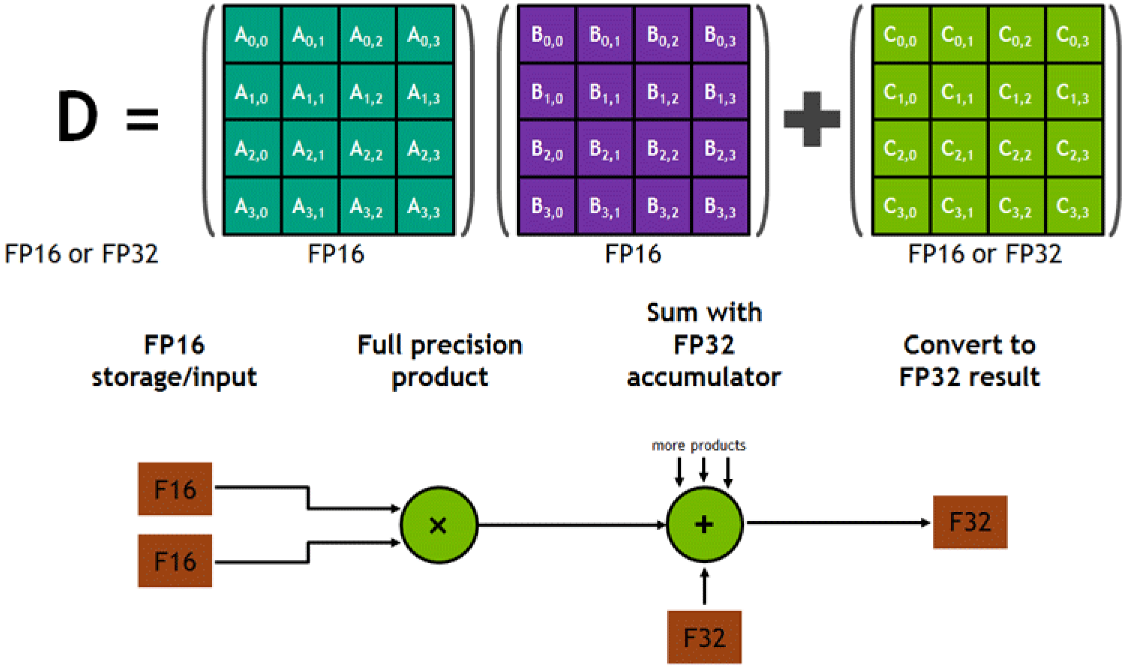
GPGPU Tensor Cores for Deep Learning (NVIDIA)

Example of NVIDIA Volta

- 64x FP32 cores per SM
- 32x FP64 cores per SM
- 8x Tensor cores per SM

Tensor core operations

- Tensor Core perform D = A x B + C, where A, B, C and D are matrices
- A and B are FP16 4x4 matrices
- D and C can be either FP16 or FP32 4x4 matrices
- Higher performance is achieved when A and B dimensions are multiples of 8
- Maximum of 64 floating-point mixed-precision FMA operations per clock



$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

Limitations of GPGPUs for Accelerated Computing

Restrictions of GPGPU programming

- CUDA is a proprietary programming environment
- OpenCL programming by writing host code and device code, then connecting them through a low-level API
- GPGPU kernel programming lacks standard features of C/C++, such as recursion or accessing a file system

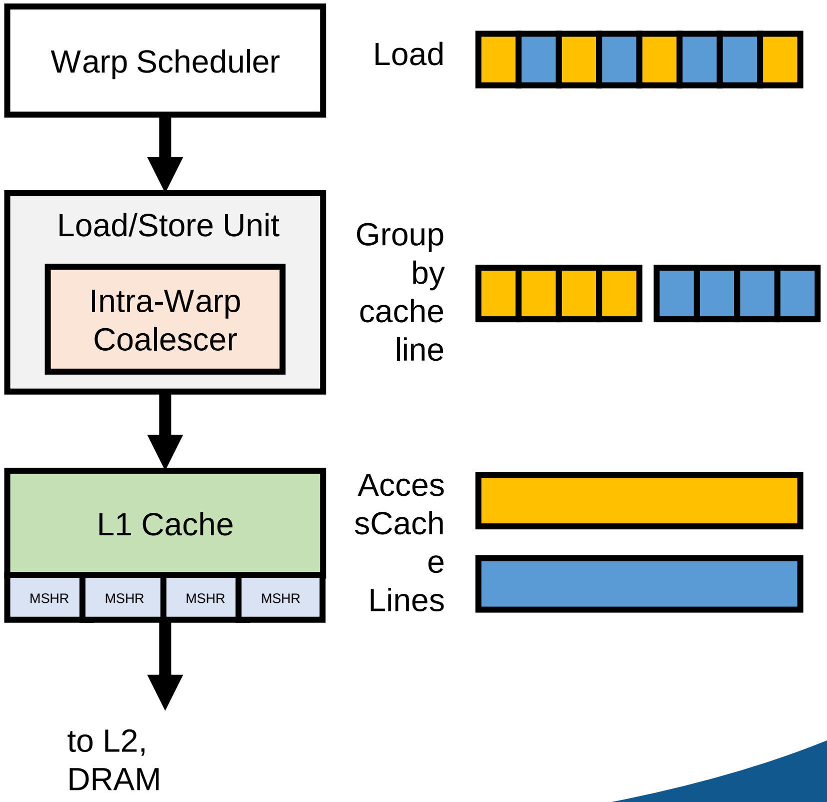
Performance issues with ‘thread divergence’

- Branch divergence: a simple if…then…else construct will force all threads in a warp to execute both the "then" and the "else" path
- Memory divergence: when hardware cannot coalesce the set of warp global memory accesses into one or two L1 cache blocks

Time-predictability issues

- Dynamic allocation of thread blocks to SMs
- Dynamic warp scheduling and out-of-order execution of warps on each SM

Memory access coalescing (Kloosterman et al.)



Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

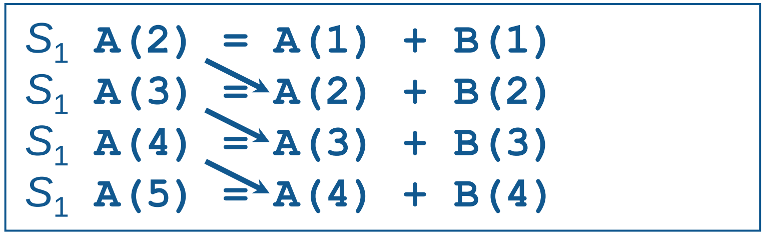
Conclusions

Data Dependences in Innermost Loops

Scalar loop

- Loop-carried dependences lexically backward or left-to-right in statement.

```
   DO I = 1, N
S1  A(I+1) = A(I) + B(I)
   ENDDO
```



S_1 A(2) = A(1) + B(1)

S_1 A(3) = A(2) + B(2)

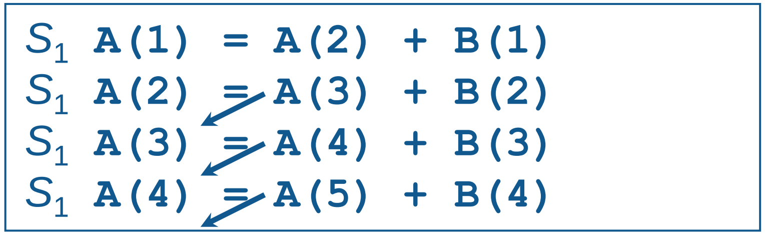
S_1 A(4) = A(3) + B(3)

S_1 A(5) = A(4) + B(4)

Vector loop

- Loop-carried dependences lexically forward or right-to-left in statement.

```
   DO I = 1, N
S1  A(I) = A(I+1) + B(I)
   ENDDO
```



S_1 A(1) = A(2) + B(1)

S_1 A(2) = A(3) + B(2)

S_1 A(3) = A(4) + B(3)

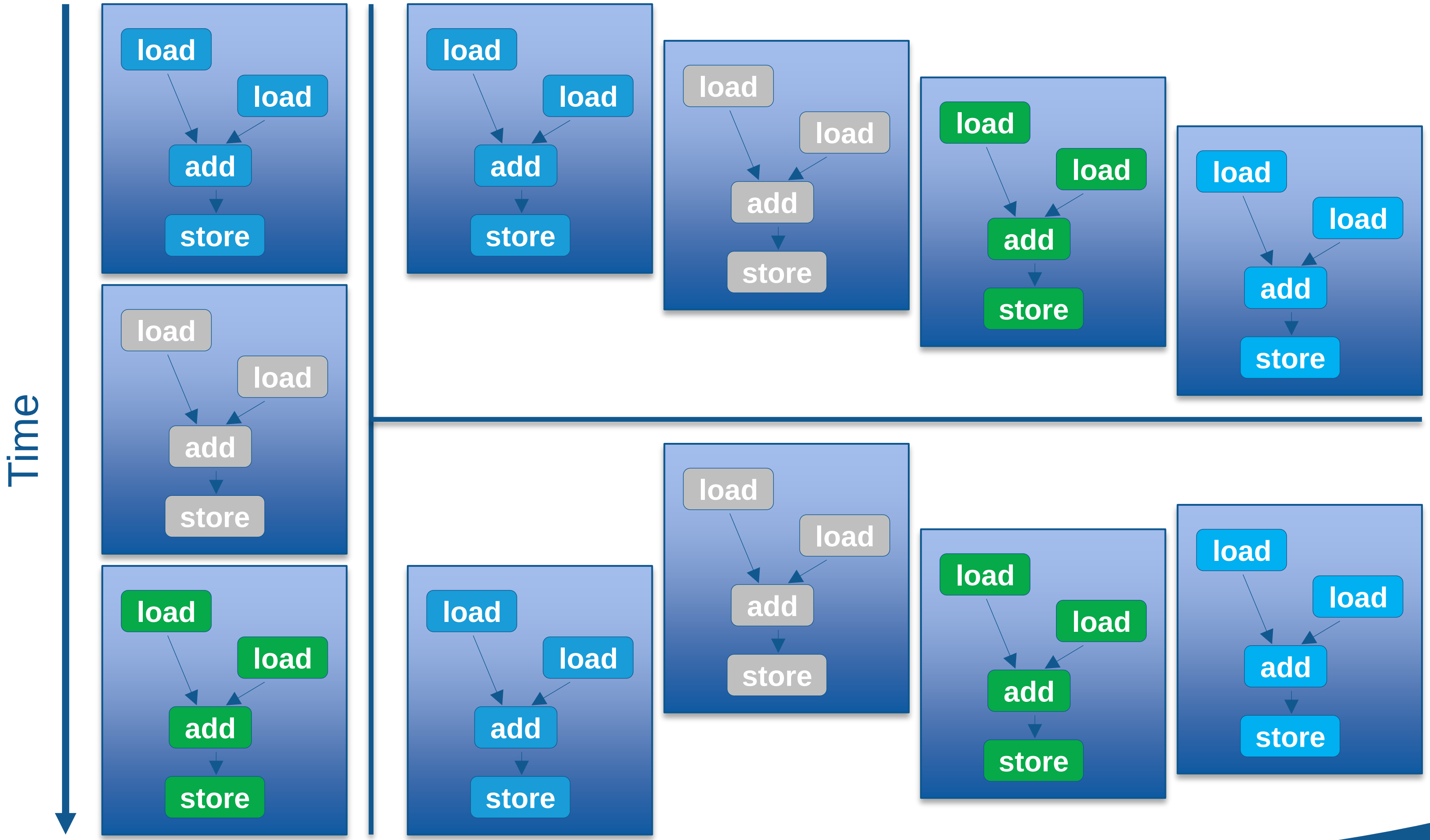
S_1 A(4) = A(5) + B(4)

Independent loop

- No loop-carried dependences

```
   DO I = 1, N
S1  A(I) = A(I) + B(I)
   ENDDO
```

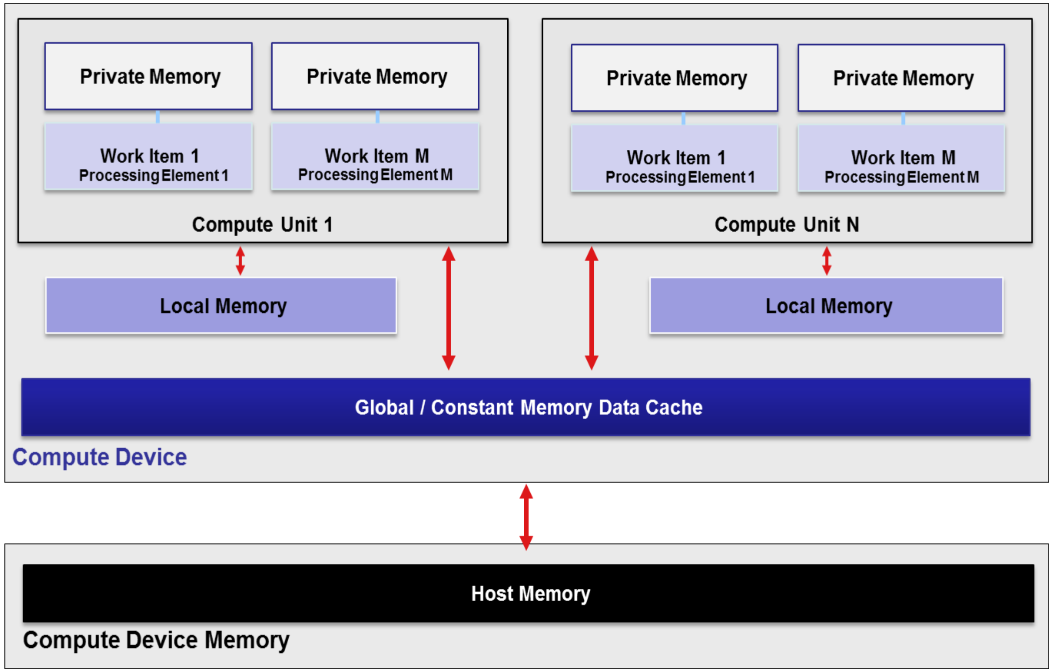
Loop Iterations: Scalar, Vector, Independent



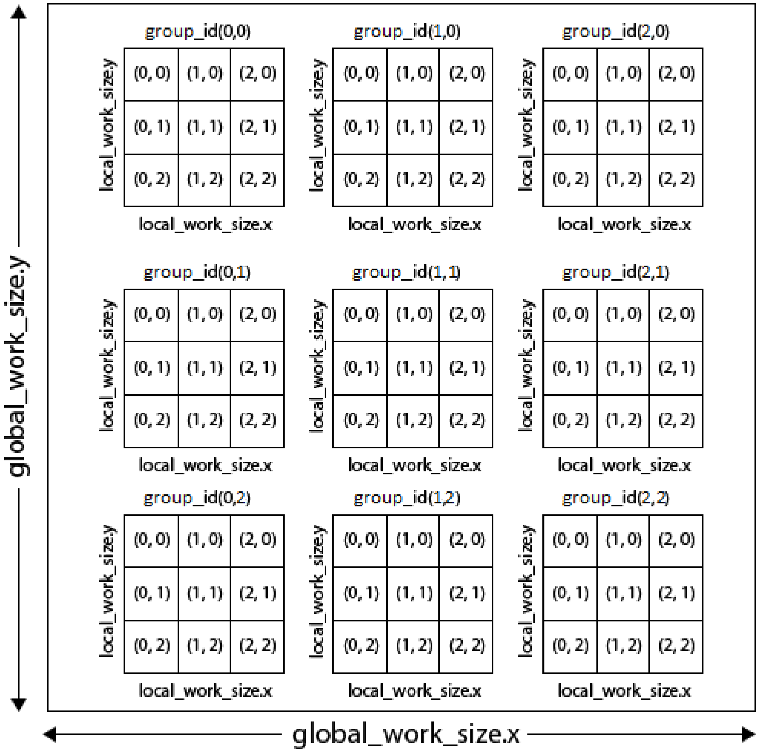
OpenCL For Manycore Processors

OpenCL 1.2 has two parallel execution models

- Data parallel, with one work item per processing element
- Task parallel, with one work item per compute unit
  - Task parallel model is exploited by FPGAs and by DSPs



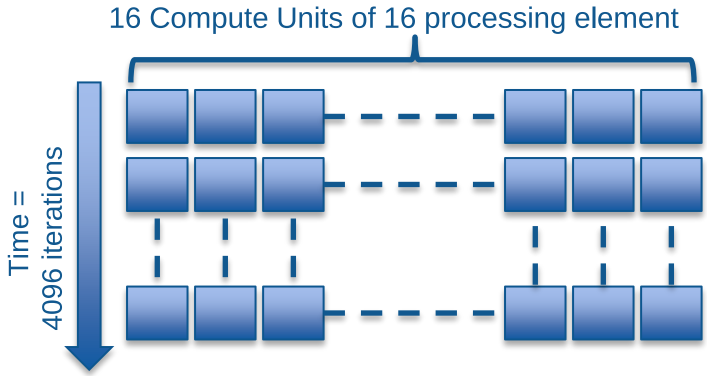
NDRange



OpenCL Data Parallel Model

Executes a kernel at each point in a grid domain

- To process a 1024 x 1024 image
- Create one kernel instance per pixel
  - 1,048,576 kernel executions
  - 65536 working groups of 16 work items
  - 4096 iterations on 16 Compute Units of 16 Processing Elements



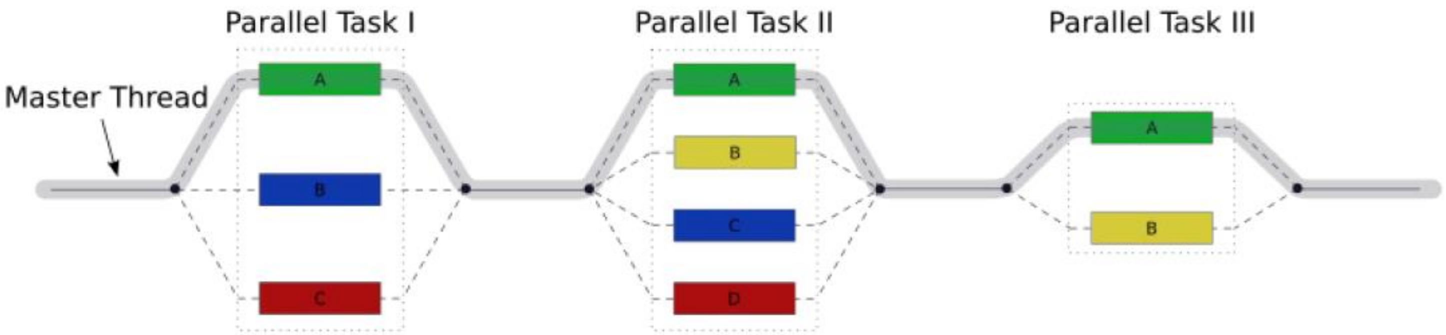
```
#define n 256

void vecAdd( const double *a,
             const double *b,
             double *c,
             int n)
{
     int i;
     for (i=0; i<n; i++) {
          c[i] = a[i] + b[i];
     }
}
```

```
__kernel void vecAdd( __constant double *a,
                      __constant double *b,
                      __global double *c)
{
     unsigned int id = get_global_id(0);
     c[i] = a[i] + b[i];
}

/* 16 work items in each workgroup */
size_t localSize = 16;
/* Start 16 work-groups of 16 work-items) */
size_t globalSize = localSize * 16;
/* enqueue the tasks 16 work-groups of 16 work-
items */
err = clEnqueueNDRangeKernel(queue, kernel, 1,
NULL, &globalSize, &localSize, 0, NULL, NULL);
```

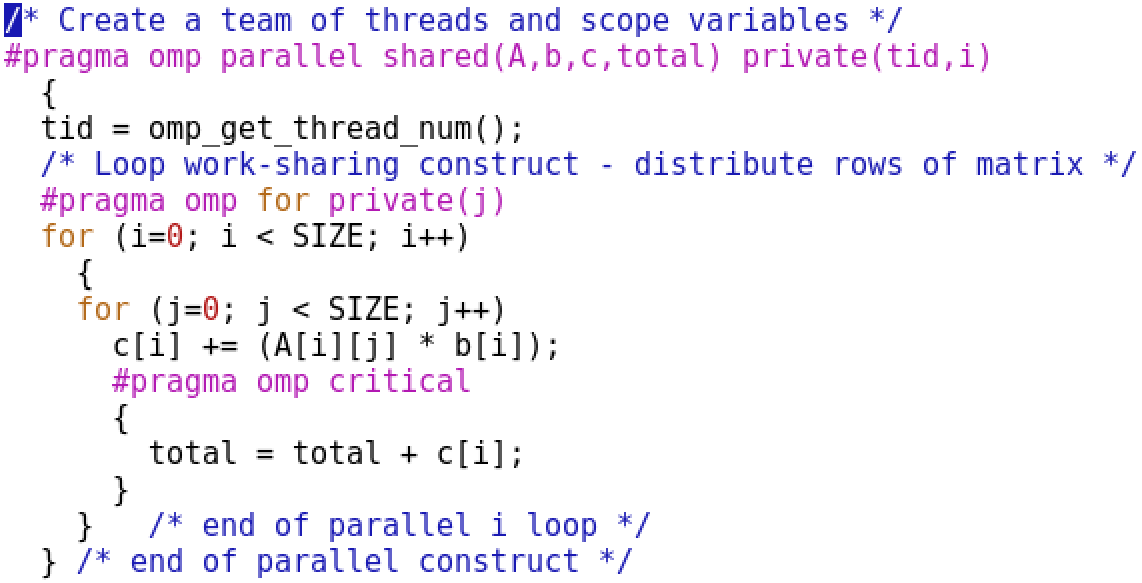
OpenMP for Multicore Programming



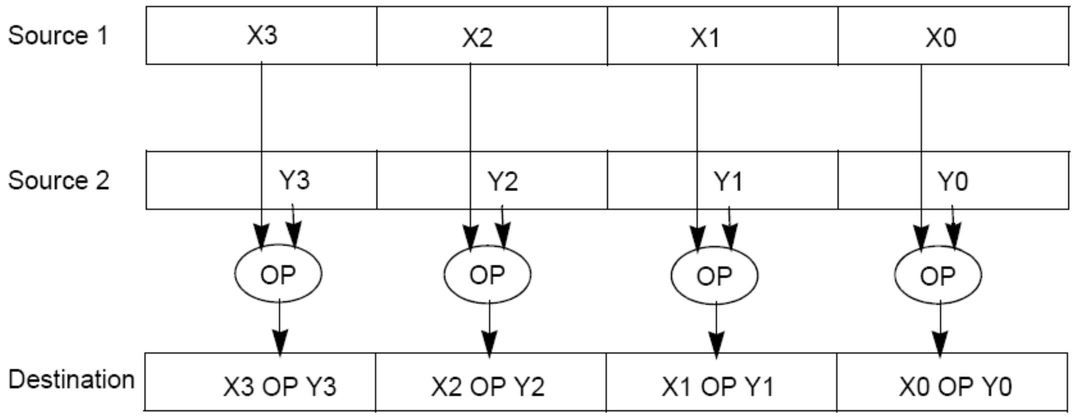
A parallel region starts redundant execution

Work sharing constructs assign different pieces of work to threads

Synchronization is explicit (here critical section) or implicit (barriers end constructs)


```
/* Create a team of threads and scope variables */
#pragma omp parallel shared(A,b,c,total) private(tid,i)
 {
 tid = omp_get_thread_num();
 /* Loop work-sharing construct - distribute rows of matrix */
 #pragma omp for private(j)
 for (i=0; i < SIZE; i++)
  {
  for (j=0; j < SIZE; j++)
   c[i] += (A[i][j] * b[i]);
   #pragma omp critical
   {
    total = total + c[i];
   }
  }   /* end of parallel i loop */
 } /* end of parallel construct */
```

OpenMP for SIMD/Vector Execution



Source 1
X3
X2
X1
X0

Source 2
Y3
Y2
Y1
Y0

OP
OP
OP
OP

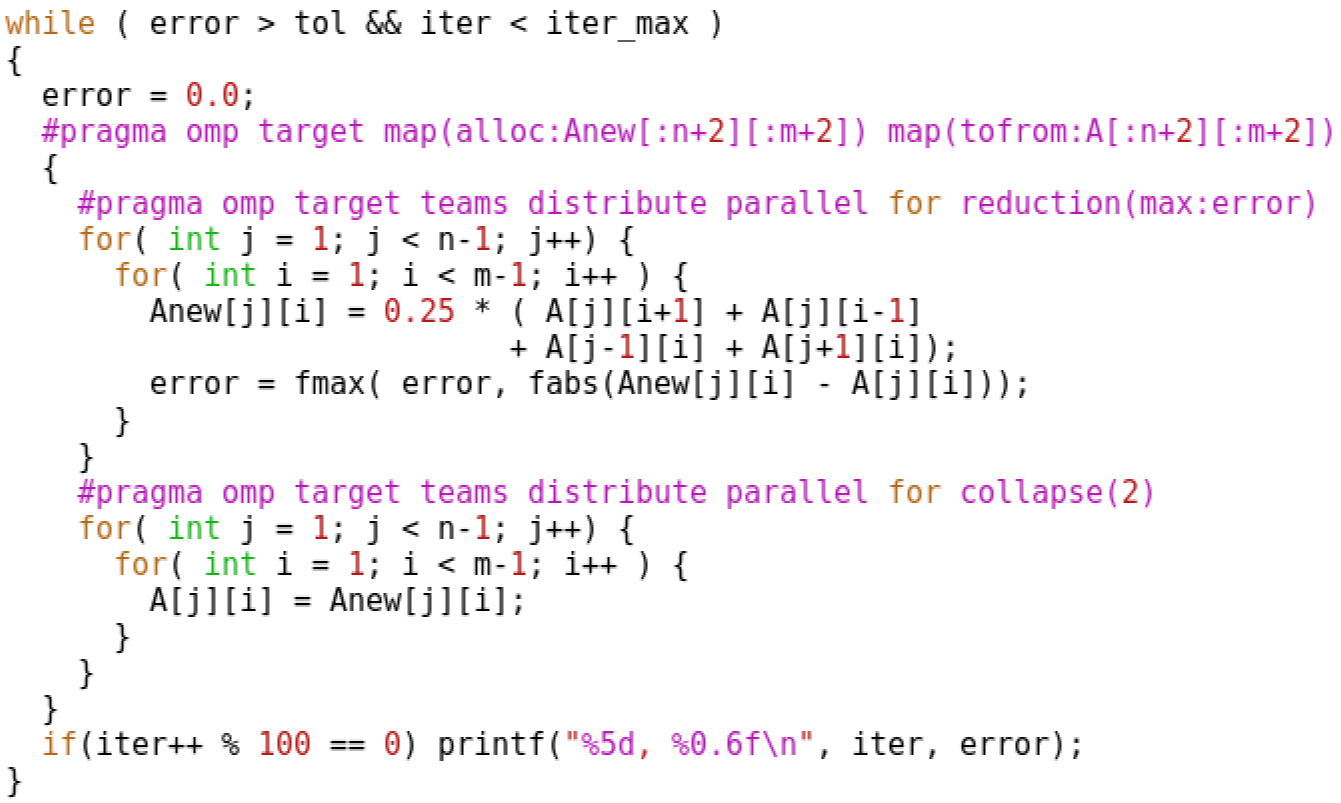
Destination
X3 OP Y3
X2 OP Y2
X1 OP Y1
X0 OP Y0

```
#pragma omp simd
{
  for (i=0; i<N; i++) {
    a[i] = a[i] + b[i] * c[i];
  }
}
```

```
float x[N][N], y[N][N];
#pragma omp parallel
{
  #pragma omp for
  for (int i=0; i<N; i++) {
    #pragma omp simd safelen(18)
    for (int j=18; j<N-18; j++) {
      x[i][j] = x[i][j-18] + sinf(y[i][j]);
      y[i][j] = y[i][j+18] + cosf(x[i][j]);
    }
  }
}
```

OpenMP for Accelerator Offloading

First map data to the accelerator, then distribute work to the accelerator threads



```
while ( error > tol && iter < iter_max )
{
  error = 0.0;
  #pragma omp target map(alloc:Anew[:n+2][:m+2]) map(tofrom:A[:n+2][:m+2])
  {
    #pragma omp target teams distribute parallel for reduction(max:error)
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                            + A[j-1][i] + A[j+1][i]);
        error = fmax( error, fabs(Anew[j][i] - A[j][i]));
      }
    }
    #pragma omp target teams distribute parallel for collapse(2)
    for( int j = 1; j < n-1; j++) {
      for( int i = 1; i < m-1; i++ ) {
        A[j][i] = Anew[j][i];
      }
    }
  }
  if(iter++ % 100 == 0) printf("%5d, %0.6f\n", iter, error);
}
```

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

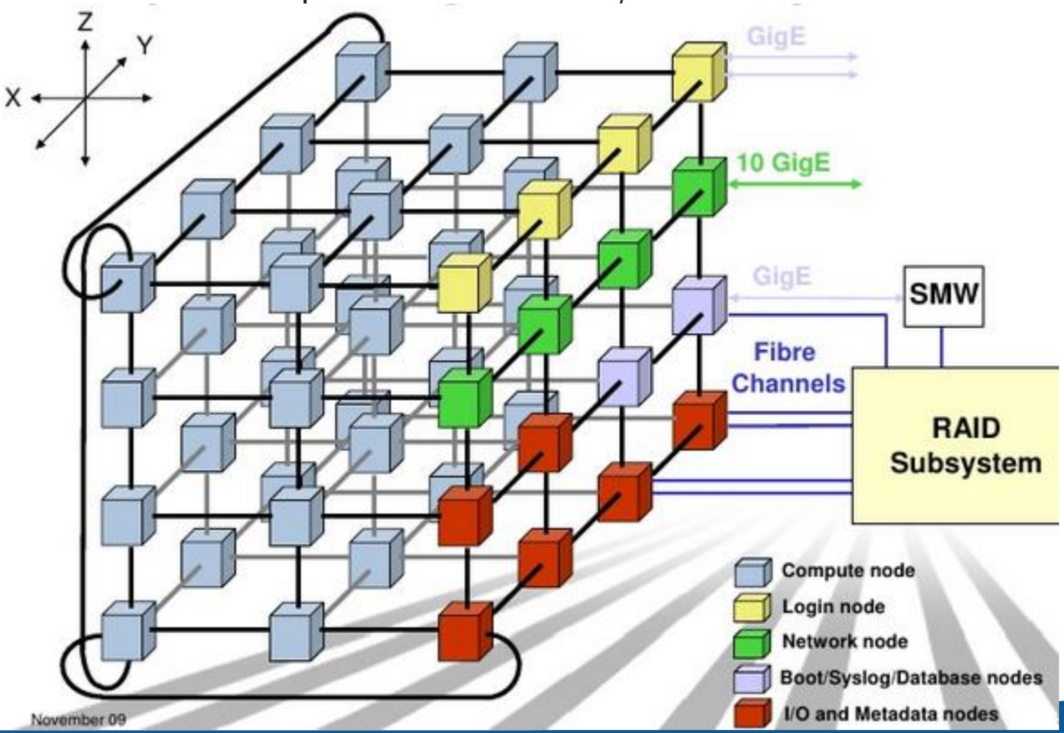
Deep Learning Inference

Conclusions

Supercomputer Distributed Memory Architecture

IBM BlueGene series, Cray XT series

- Compute nodes with multiple cores and shared memory
- I/O nodes with high-speed devices and a Linux operating system
- Specialized networks between the compute nodes and the I/O nodes



What is Symmetric Parallel Programming

A Single Program Multiple Data (SPMD) execution model

- All processes execute the same code
- Processes participate in collective operations
- Global data is seen as the combination of local data
- One-sided communications and bulk synchronizations

Variants of symmetric parallel programming models

- Gorlatch-style MPI programming ‘Send-Receive Considered Harmful’
- Supercomputer communication libraries:
  - Cray SHMEM, DoE ARMCI, Berkeley GASNet, IBM DCMF & PAMI
- Partitioned Global Address Space (PGAS) languages:
  - Co-Array Fortran (CAF), Unified Parallel C (UPC), Titanium
- Bulk Synchronous Parallel (BSP) programming models

Cray SHMEM Communication Library (1995)

Origin and uses

- Introduced by Cray (1993) for the Cray T3D
- Supported by SGI (1997), Quadrics (1998)
- GPSHMEM (2000) implementation on top of ARMCI
- Base of Cray F-- (1997), which became co-array Fortran
- Evolutions: ordered -> unordered, blocking -> non blocking

One-sided primitives + atomic, collective operations

- `shmem_long_put(dst, src, len, pe); shmem_long_get(...);`
- `shmem_swap(dst, src, pe); shmem_wait(var, value);`
- `shmem_long_sum_to_all(...);`
- `shmem_barrier(...); shmem_fence(); shmem_quiet();`

Symmetric memory allocation

- Replicated static variables at same local address
- Dynamic memory allocation: shmalloc(size);

Co-Array Fortran for Distributed Memory (2008)

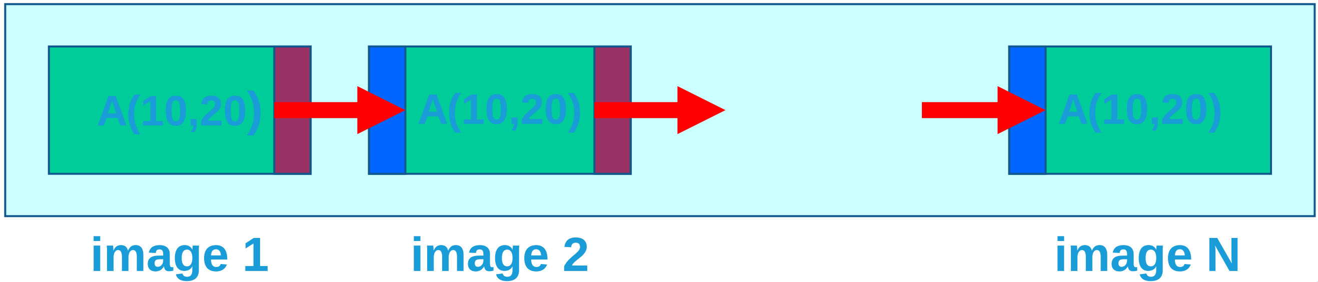
Simple Program Multiple Data with symmetric data

- Same program replicated as a fixed number of concurrent ‘images’
- Images execute asynchronous ‘segments’ between SYNC statements
- Global data is composed of image data with co-dimensions
  - `FLOAT A(10,20)[*]`

Co-Array Fortran syntax extensions

- Uses normal rounded brackets ( ) to point to data in local memory
- Uses square brackets [ ] to point to data in remote memory

  -
    ```
    IF (this_image() > 1) ! Get data from left neighbor
    A(1:10,1:2) = A(1:10,19:20)[this_image()-1]
    ```



Bulk Synchronous Parallel Models

The ‘bridging model’ of L. Valiant

- SPMD and distributed memory
- Bulk message passing
- Superstep synchronization

Oxford University BSPLib

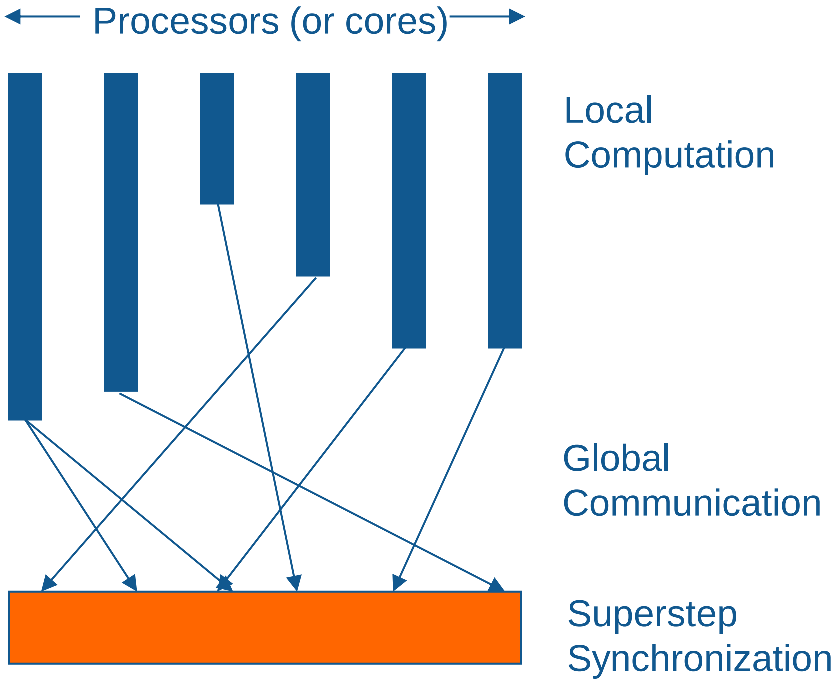
- Introduce put() and get() one-sided operations like SHMEM
- Registration of local objects <=> symmetric memory space

Paderborn University BSP (PUB)

- Collectives (reduce, scan, etc.)
- Split BSP to sub-BSP machines

NestStep languages of C. Kessler

- Manage ‘replicated’ and ‘distributed’ objects



effects of communications visible only after superstep

Minimal BSP Interface

SPMD image queries

- `bsp_order(); // Number of images in the SPMD program`
- `bsp_rank(); // Rank of image, in [0 … bsp_order()-1]`

Registration and bulk synchronization

- `bsp_register(object, size); // Register  a local object for communications`
- `bsp_unregister(n); // Undo n latest calls to bsp_register()`
- `bsp_sync(); // Superstep synchronization`

One-sided communications

- `bsp_put(rank, object, addr, data, size); // Put data to registered object`
- `bsp_get(rank, object, addr, data, size); // Get data from registered object`

Delayed communications semantics

- While executing superstep, capture put() sources in buffers
- At the end of superstep, capture get() sources in registered data
- Write data associated with put() or get() to destinations

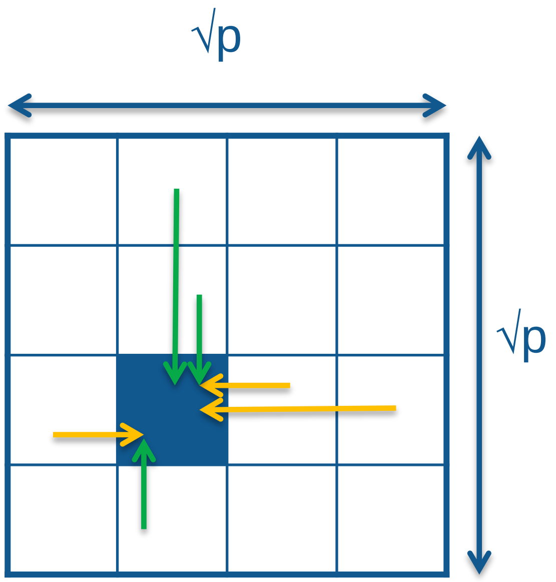
Dense Matrix Multiplication with the BSP Model

Dense matrix-matrix multiplication algorithm by Gerbessiotis

- P images, each image maintains a tile of matrices A, B, C
- Each image receives other tiles by ‘get’ operations followed by ‘sync’

Gerbessiotis MatMulG algorithm

```
Let $q = \text{pid}$
Let $p_i = q \mod \sqrt{p}$
Let $p_j = q/\sqrt{p}$
Let $C_q = 0$
for $0 \leq l < \sqrt{p}$ do
    $a \leftarrow A_{((p_i+p_j+l) \bmod \sqrt{p})*\sqrt{p}+p_i}$
    $b \leftarrow B_{((p_i+p_j+l) \bmod \sqrt{p})+\sqrt{p}*p_j}$
    sync
    Let $b^t = \text{transpose}(b)$
    $C_q += a \times^t b^t$
end for
```

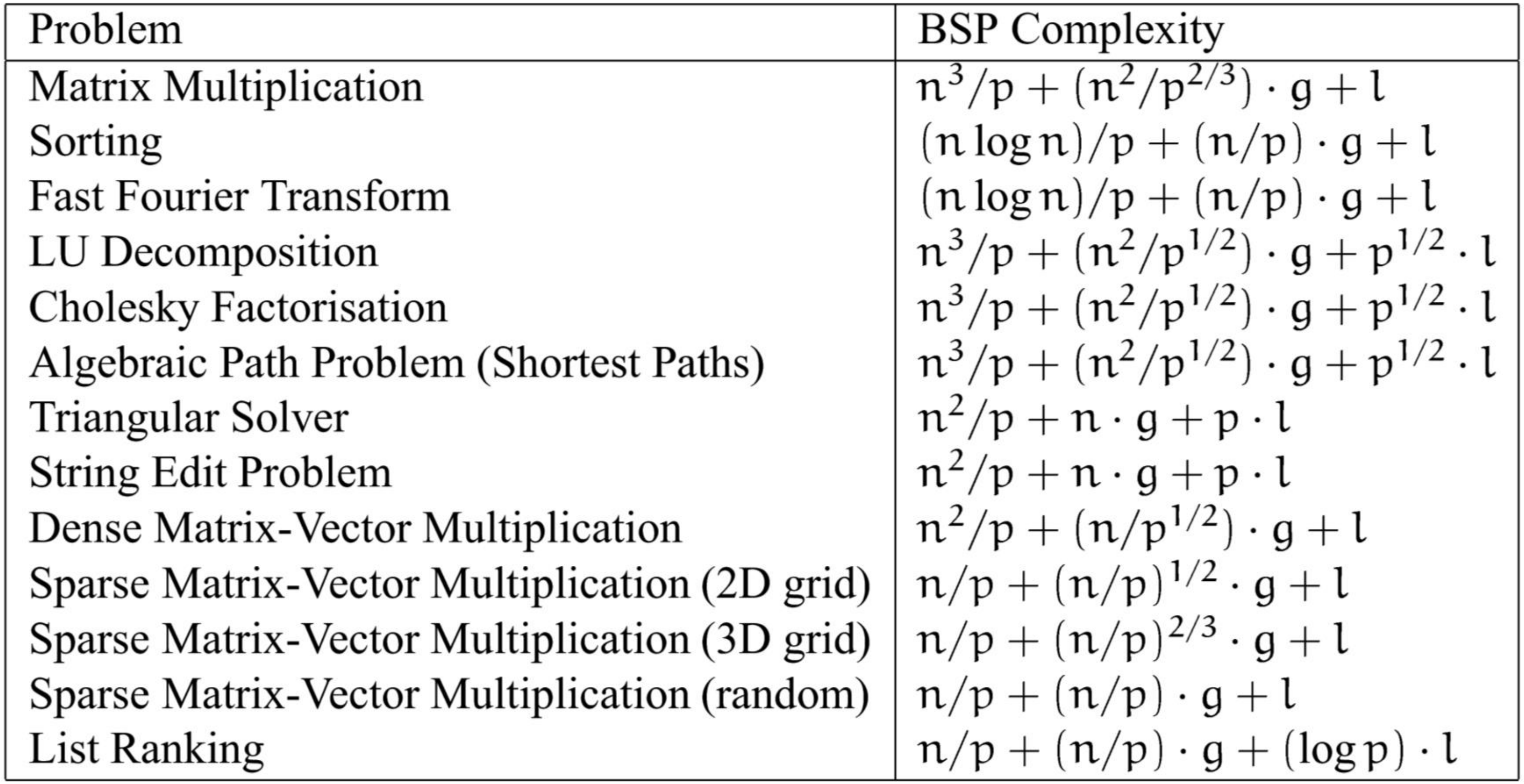


← get from matrix A

← get from matrix B

Distributed Algorithms with the BSP Model

McColl 1998 “Foundations of Time-Critical Scalable Computing”



Problem	BSP Complexity
Matrix Multiplication	$n^3/p + (n^2/p^{2/3}) \cdot g + l$
Sorting	$(n \log n)/p + (n/p) \cdot g + l$
Fast Fourier Transform	$(n \log n)/p + (n/p) \cdot g + l$
LU Decomposition	$n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$
Cholesky Factorisation	$n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$
Algebraic Path Problem (Shortest Paths)	$n^3/p + (n^2/p^{1/2}) \cdot g + p^{1/2} \cdot l$
Triangular Solver	$n^2/p + n \cdot g + p \cdot l$
String Edit Problem	$n^2/p + n \cdot g + p \cdot l$
Dense Matrix-Vector Multiplication	$n^2/p + (n/p^{1/2}) \cdot g + l$
Sparse Matrix-Vector Multiplication (2D grid)	$n/p + (n/p)^{1/2} \cdot g + l$
Sparse Matrix-Vector Multiplication (3D grid)	$n/p + (n/p)^{2/3} \cdot g + l$
Sparse Matrix-Vector Multiplication (random)	$n/p + (n/p) \cdot g + l$
List Ranking	$n/p + (n/p) \cdot g + (\log p) \cdot l$

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

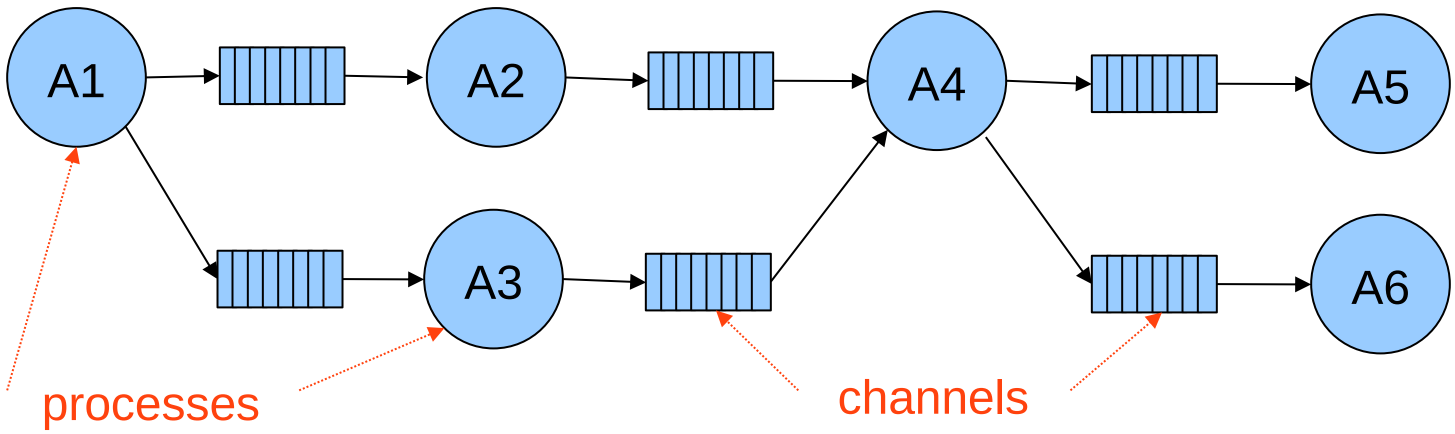
Kahn Process Networks (KPNs) [Kahn 1974]

Sequential “processes” connected through FIFO “channels”

Blocking “read”, non blocking “write” on channels

Processes are also called “actors” or “agents”

Determinacy of results, independent of actor firing sequence



Dataflow Models of Computation

Dataflow Process Networks (DPN) [Lee & Parks 1995]

- Kahn Process Network with functional actors (no persistent state) and sequential firing rules (pre-defined order using only blocking reads)

Static Dataflow (SDF) [Lee & Messerschmitt 1987]

- Agents producing and consuming a constant number of tokens
- Single-rate SDF is also known as Homogenous SDF (HSDF)

Synchronous Dataflow (SDF) [Benveniste et al. 1994]

- Time advances in lockstep with one or more clocks (Signal, Esterel, Lustre, SCADE Suite)

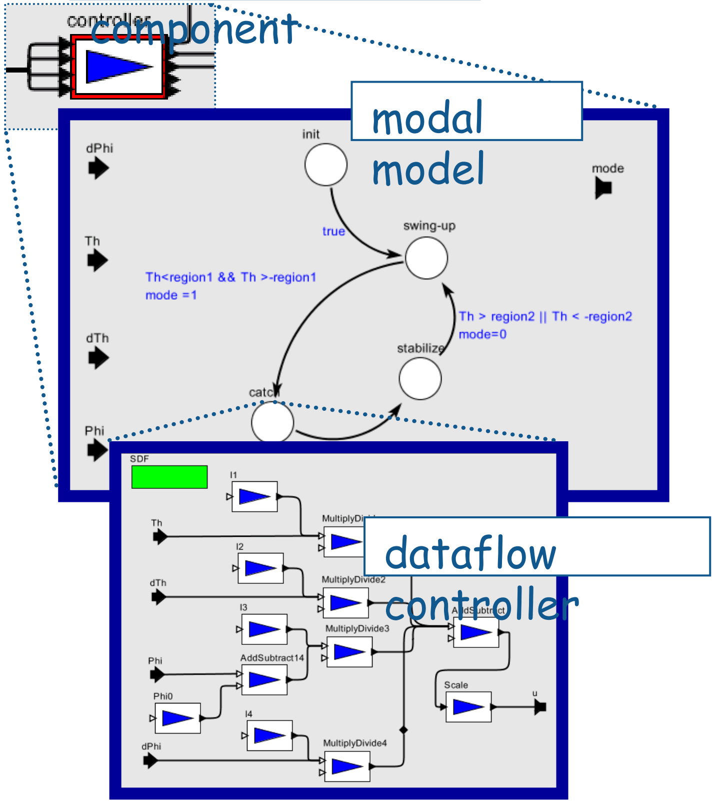
Cyclo-Static Dataflow (CSDF) [Lauwereins 1994]

- A cyclic state machine unconditionally advances at each firing
- Known number of tokens produced and consumed for each state

Computational Process Networks (CPN) [Karp & Miller 1966]

- SDF extended with ‘firing thresholds’: # input tokens > # consumed tokens

Ptolemy II (Berkeley) for Actor-Oriented Design



Hierarchical component

modal model

dataflow controller

example Ptolemy II model: hybrid control system

Framework for experimentation with actor-oriented design, concurrent semantics, visual syntaxes, and hierarchical, heterogeneous design.

http://ptolemy.eecs.berkeley.edu

StreamIt

http://cag.lcs.mit.edu/streamit

Filters are unit of computation

- No global resources

FIFO channels operations

- peek(index) / pop() / push(value)
- peek / pop / push rates must be constant

Graph optimizations

- Horizontal/vertical filter fusion/fission
- Time/frequency domains

Teleport messaging

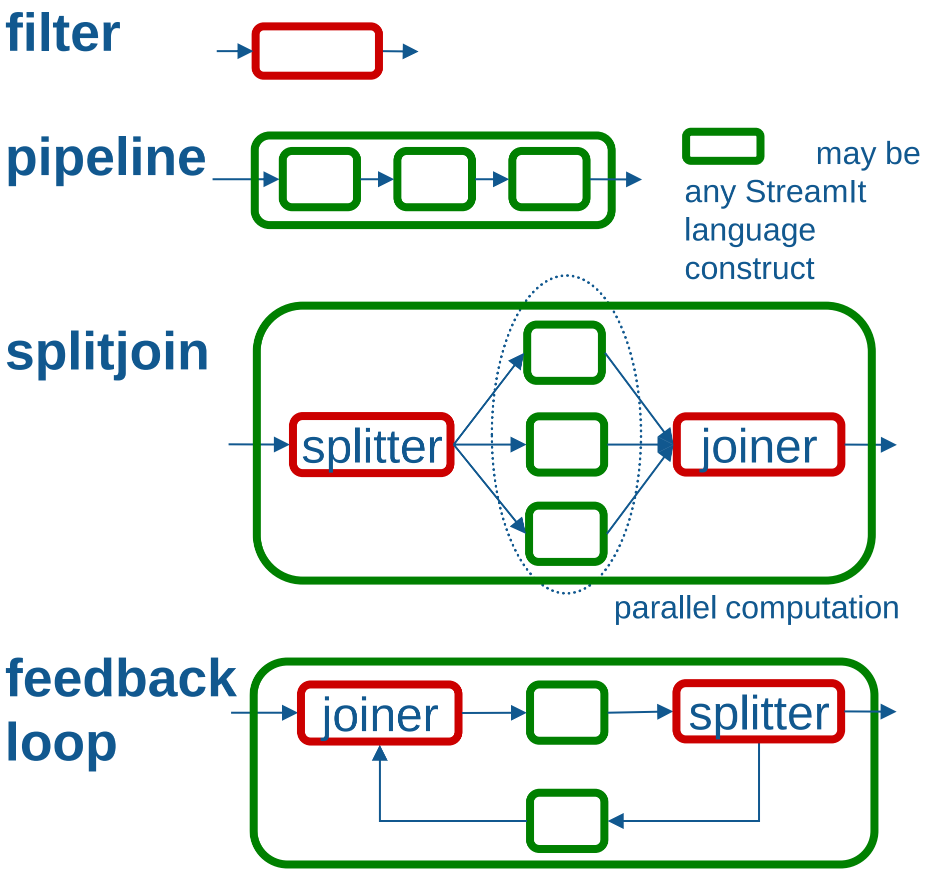
- Synchronize mode changes with data flow

Program morphing

- Update application graph while running

RAW machine code generation

- RAW project founded Tilera



filter

pipeline

may be any StreamIt language construct

splitjoin

splitter

joiner

parallel computation

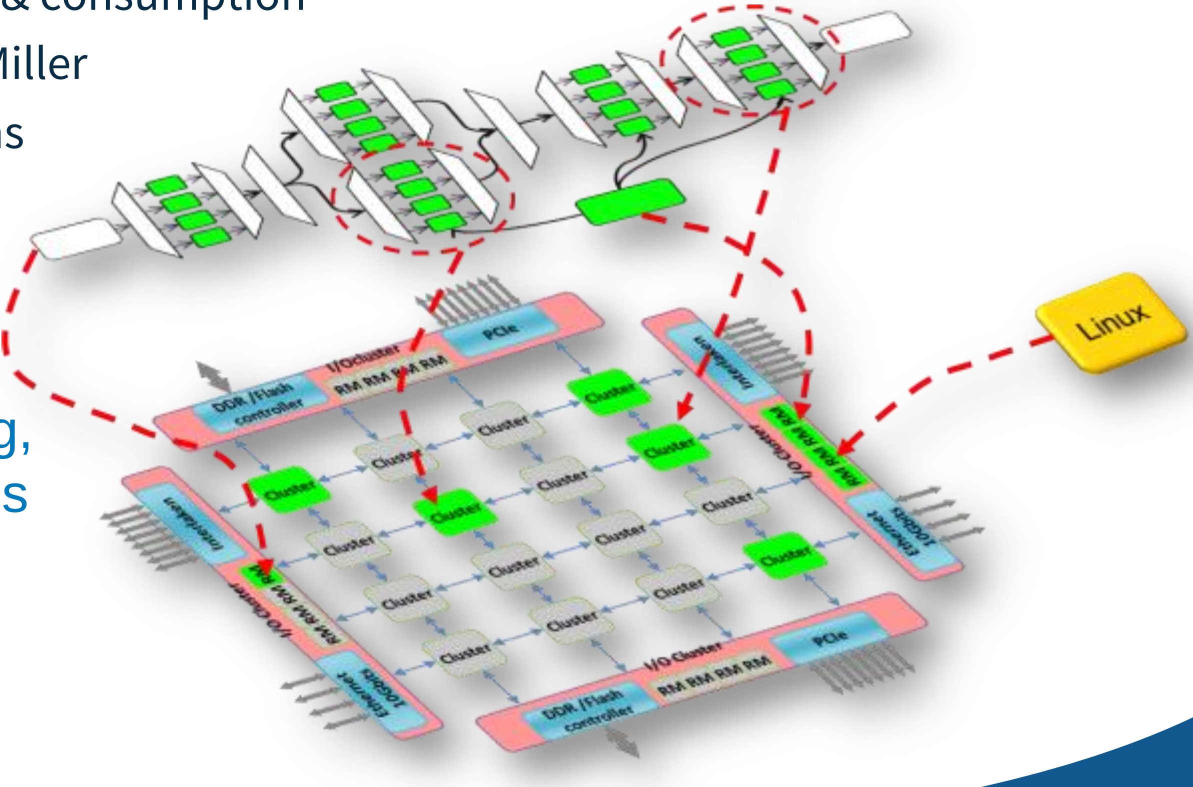
feedback loop

joiner

splitter

Sigma-C Dataflow Programming Environment

- Computation blocks and communication graph written in C
- Cyclostatic data production & consumption
- Firing thresholds of Karp & Miller
- Dynamic dataflow extensions



Automatic mapping on MPPA® memory, computing, & communication resources

Sigma-C Agent Example

```
agent Inverter()
{
    interface
    {
        in<unsigned char> input; /*< input byte stream */
        out<unsigned char> output; /*< output byte stream */

        spec{input; output};
    }

    void invert (void) exchange (input pel_in, output pel_out)
    {
            pel_out = 255 - pel_in;
    }

    void start ()
    {
            invert();
    }
}
```

agent keyword followed by the name of the agent

interface section for input/ output channels

state machine specification for data production & consumption

exchange keyword flags direct operations on input / output channels

standard C code within the agent

start function is an infinite loop

Example of Cyclostatic Specs

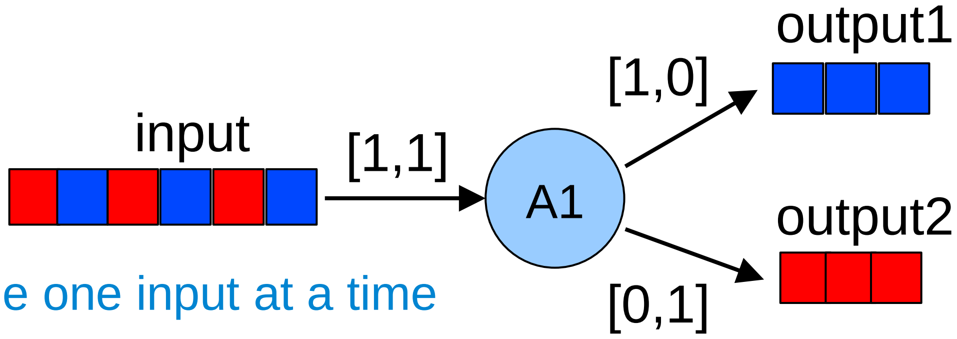
```
spec{{input; output1}; {input; output2}};

void fn1 (void) exchange (input i, output1 o)
{
        /* Function code */
}

void fn2 (void) exchange (input i, output2 o)
{
        /* Function code */
}

void start ()
{
        fn1();
        fn2();
}
```

Two exchange functions, one for each spec state



Consume one input at a time

Send every other input to a different output (split)

Generalization of Karp & Miller Thresholds

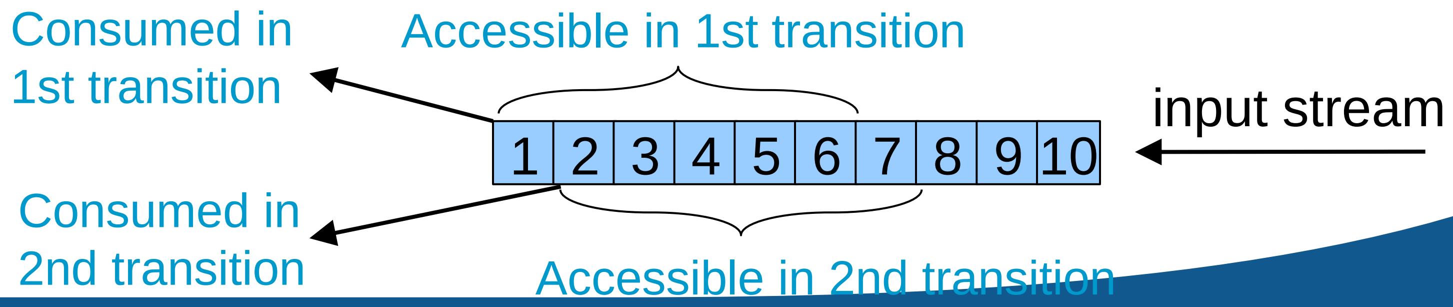
```
agent Filter()
{

   interface
   {
      in<unsigned char> input;
      out<unsigned char> output;

      spec{ {input[1:5]; output} };
   }

   void
   start (void) exchange (input i[1:5], output o)
   {
                  o = (i[0] + i[1] + i[2] + i[3] + i[4] + i[5])/3;
   }
}
```

Agent can access 6 tokens for reading but only 1 token is consumed at each transition



Consumed in 1st transition

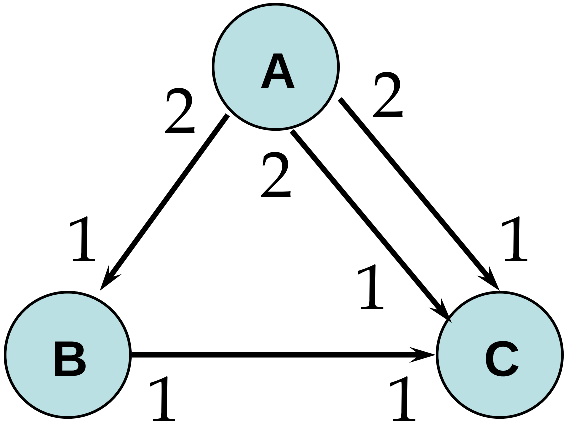
Accessible in 1st transition

input stream

Consumed in 2nd transition

Accessible in 2nd transition

Static Dataflow Graph Boundedness



Graph incidence matrix

$$M = \begin{vmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \\ 2 & 0 & -1 \\ 2 & 0 & -1 \end{vmatrix}$$

Balance equations

- 2 N(A) - N(B) = 0
- N(B) - N(C) = 0
- 2 N(A) - N(C) = 0
- 2 N(A) - N(C) = 0

Matrix must be non-full rank

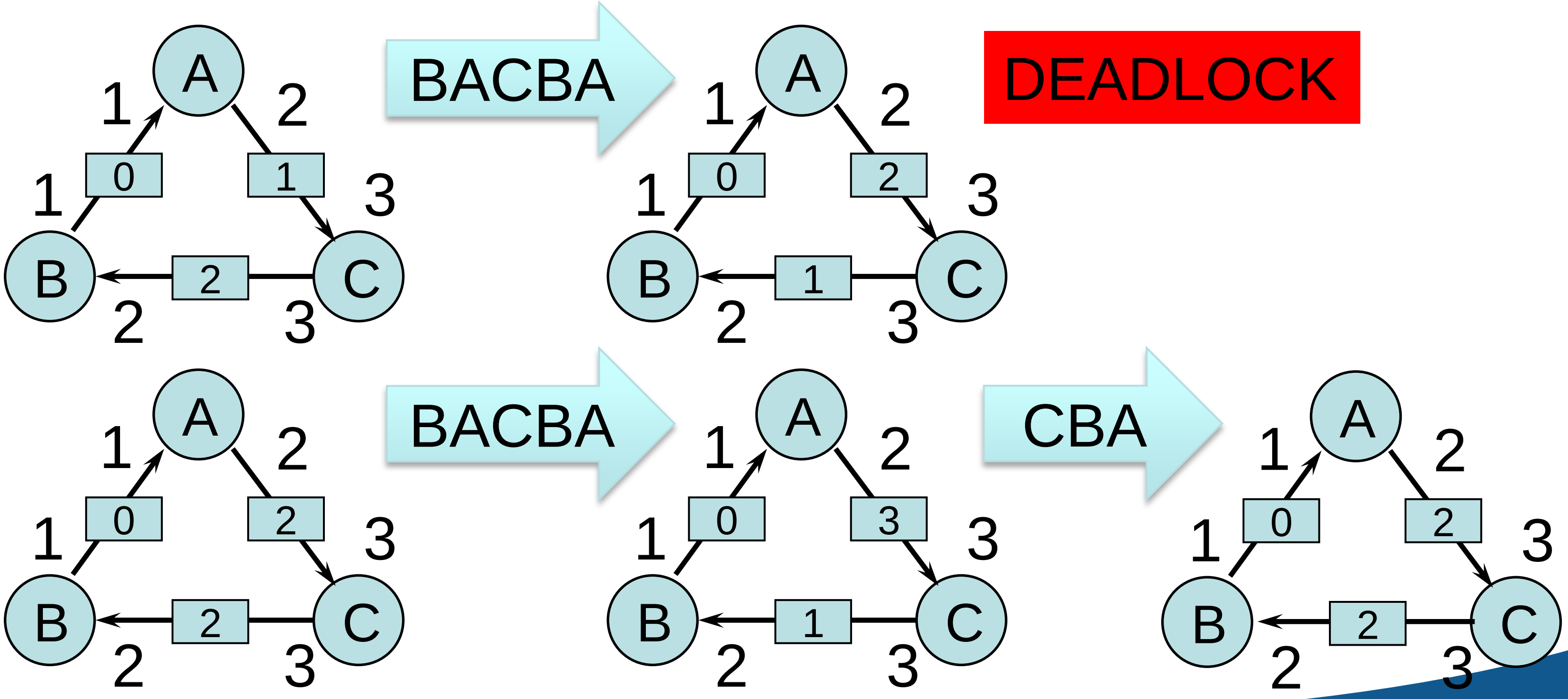
- Any multiple of the repetition vector N = |1 2 2|T satisfies the balance equations

Solution to balance equations ensures bounded buffers execution

Pre-Loaded Tokens in Channels

At program startup, some channels may be non-empty

- Required for the liveness of some dataflow graphs
- `void preload(input_channel, int token_nbr, int data_size, void *input_data);`



BACBA

DEADLOCK

BACBA

CBA

Sequencing Static Dataflow Graphs

Symbolic execution of the dataflow graph

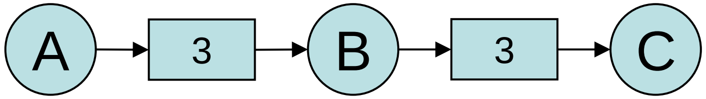
- Execute one agent firing at a time
- Find an 'hyperperiod', where each agent executes its number of times in the repetition vector and where the channel token count returns to the same values
- Preloaded tokens in channels and firing thresholds may delay the first occurrence of the hyperperiod

Symbolic execution of a balanced static dataflow graph always succeeds, unless the graph is not alive

- Take advantage of choice over ready agent firing to heuristically optimize objectives such as maximum buffer use

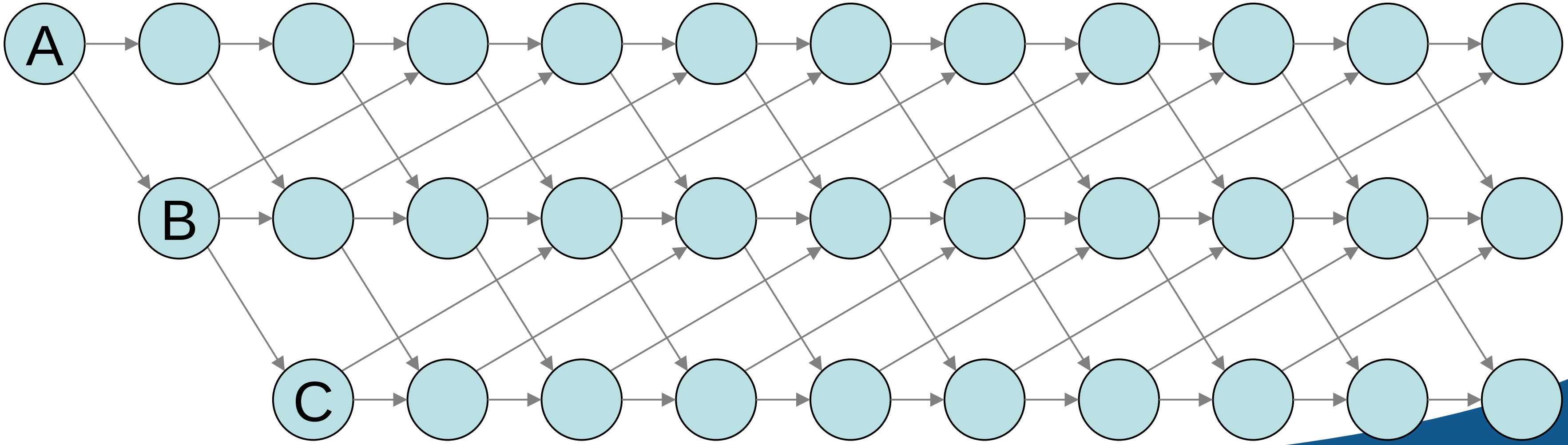
Dataflow Graph and Dependence Graph

Static Dataflow graph execution can be interpreted

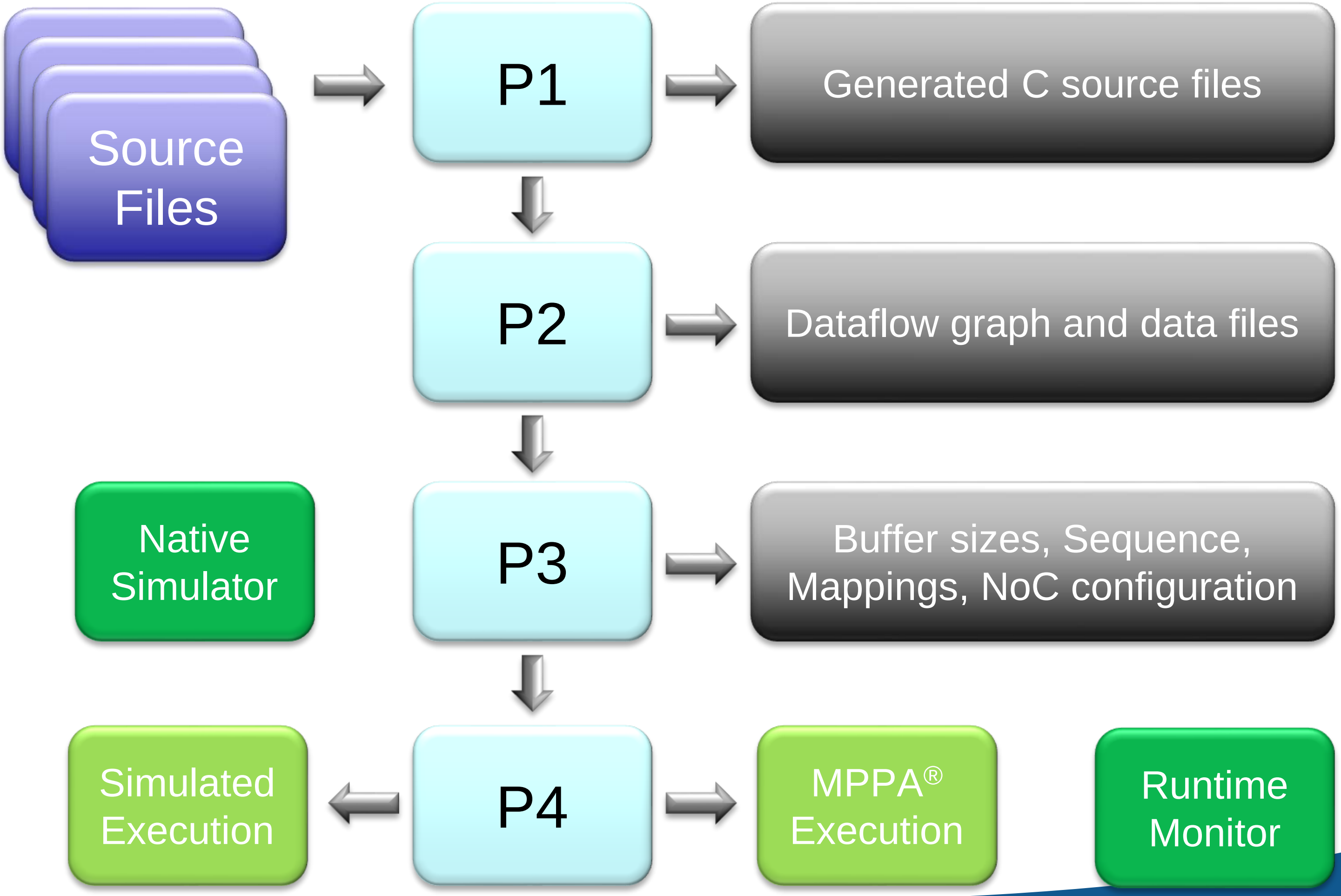


Efficient parallel execution is achieved by unfolding a dependence graph that ensures correct buffer accesses

- True data dependence arcs and buffer size feedback arcs



Sigma-C Dataflow Compilation and Execution [Retired]



Source Files

P1

Generated C source files

P2

Dataflow graph and data files

Native Simulator

P3

Buffer sizes, Sequence, Mappings, NoC configuration

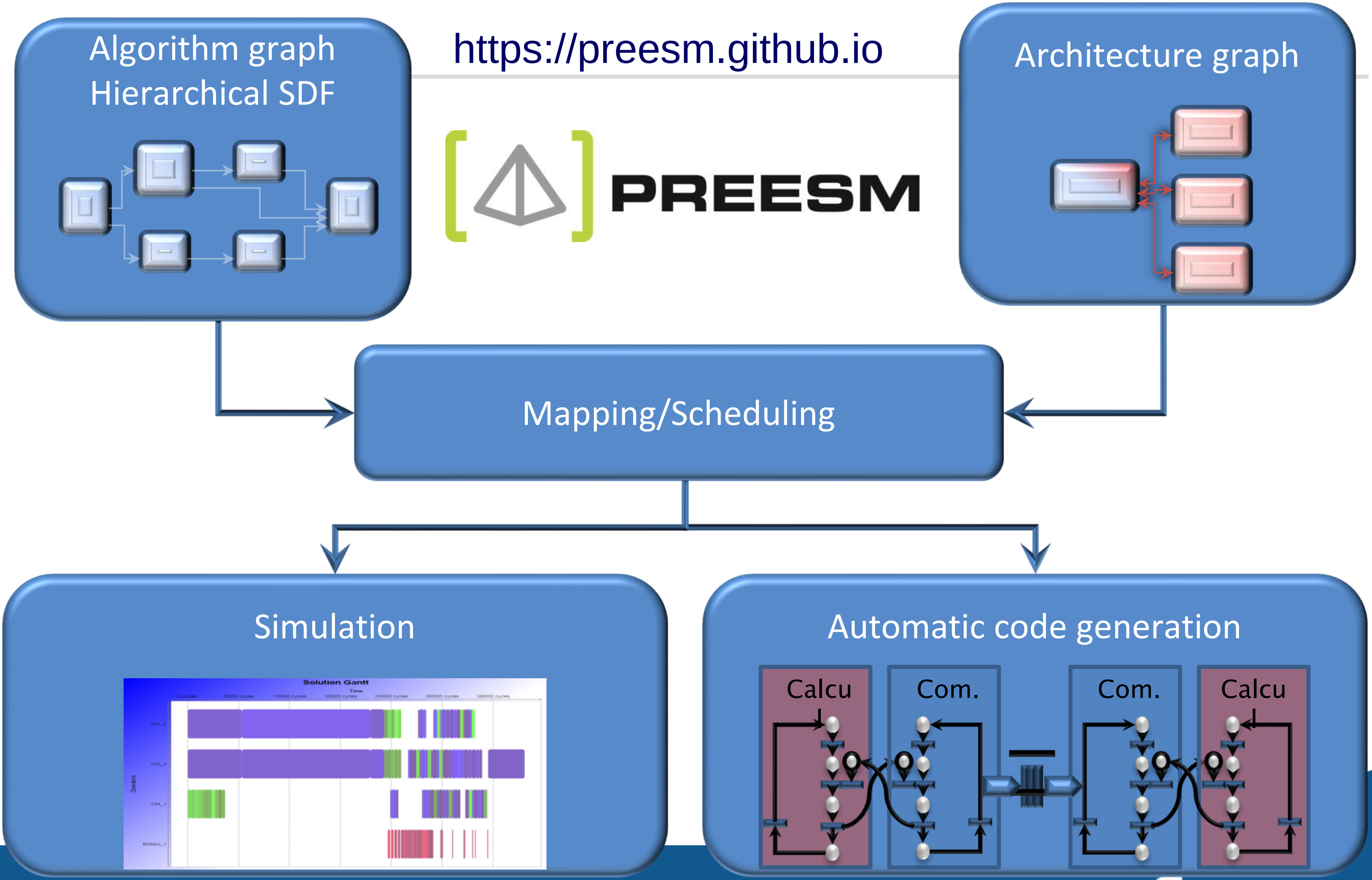
Simulated Execution

P4

MPPA® Execution

Runtime Monitor

COMPA (IETR INSA Rennes)



Algorithm graph
Hierarchical SDF

https://preesm.github.io

PREESM

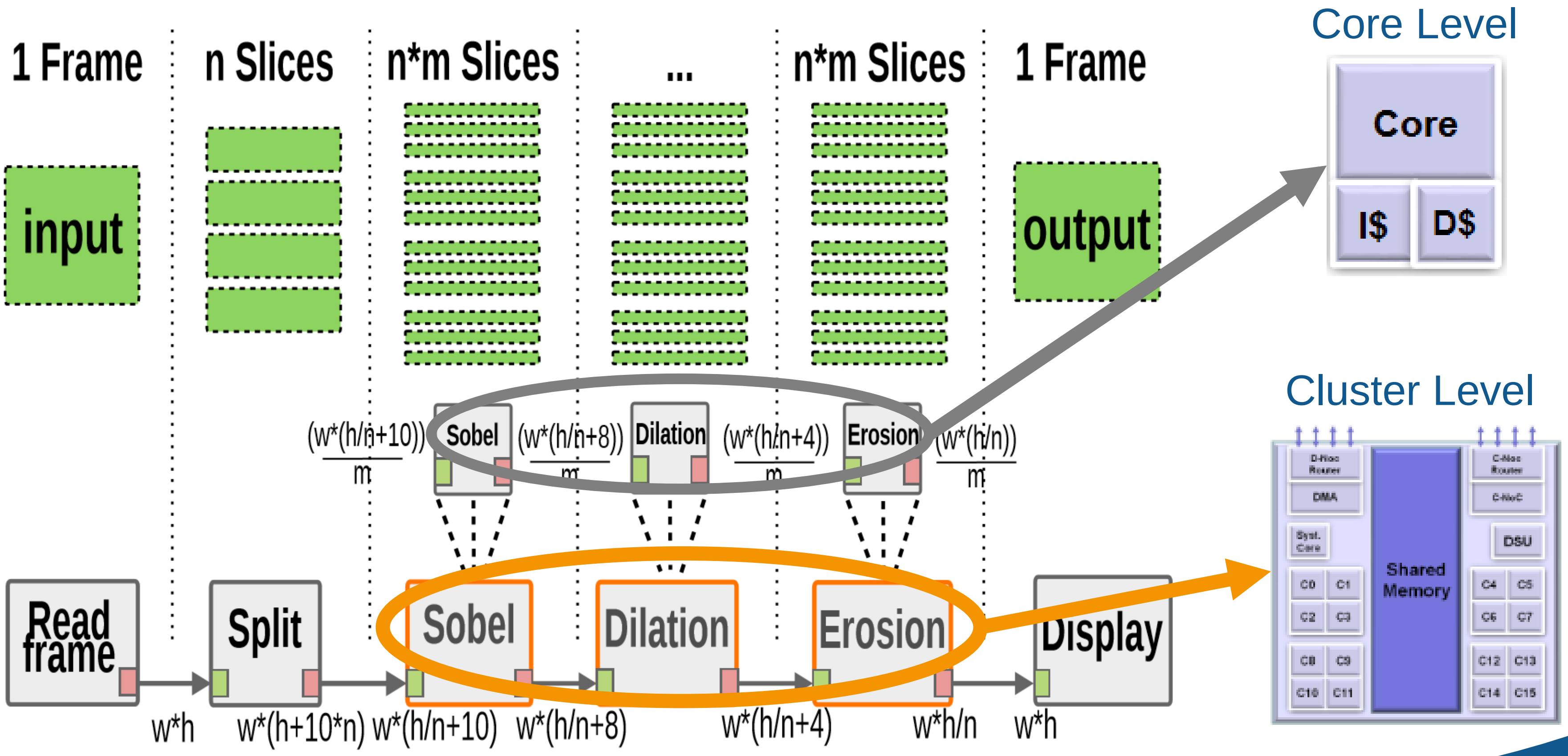
Architecture graph

Mapping/Scheduling

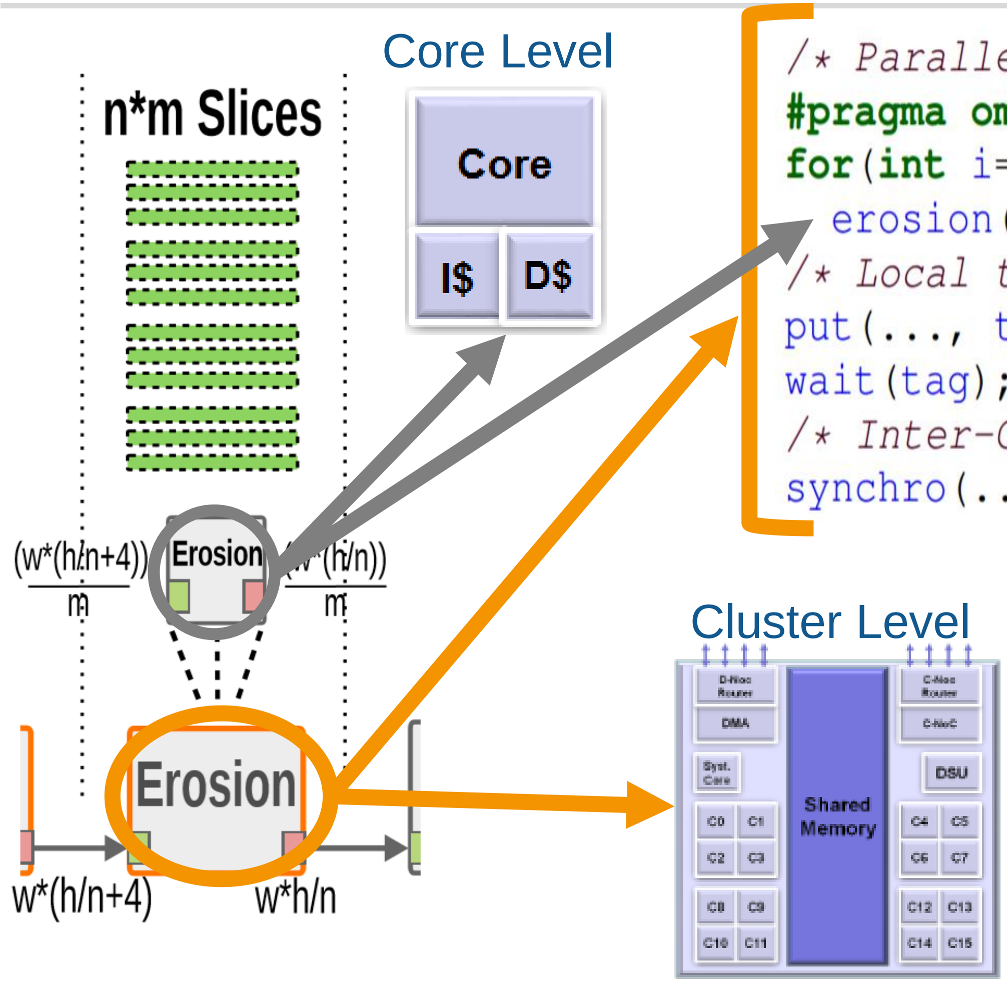
Simulation

Automatic code generation

Low-Level Image Filtering Application



Software Synthesis for MPPA® (multi-CPU model)



```
/* Parallel Hierarchical Erosion */
#pragma omp parallel for /* intra-cluster */
for(int i=0;i<M;i++)
  erosion(...); /* Erosion Kernel */
/* Local to Global Memory Coalescing */
put(...., tag); /* send buffer */
wait(tag); /* wait end of transfer */
/* Inter-Cluster Synchronization */
synchro(...);
```

OpenMP®

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

MPPA® MANYCORE HIGHLIGHTS

DSP TYPE OF ACCELERATION

- Energy efficiency
- Timing predictability
- Software programmability

CPU EASE OF PROGRAMMING

- C/C++ GNU programming environment
- Support OpenCL
- 64-bit addresses, little-endian
- Rich operating system environment

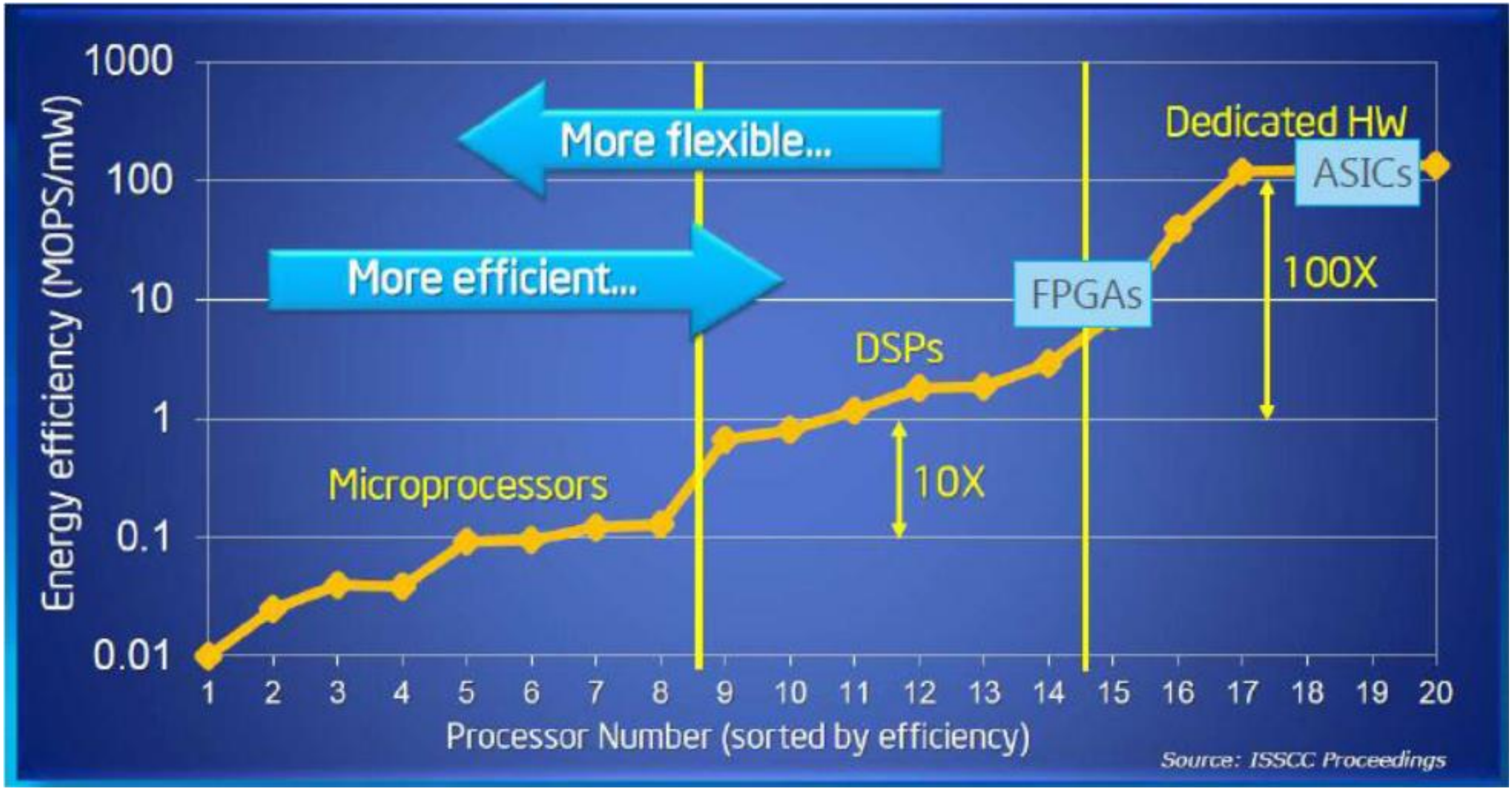
INTEGRATED MANYCORE PROCESSOR

- Secure Cores
- Application Cores
- Visio/Deep learning Co-processor
- High-performance low-latency I/O

SCALABLE MASSIVELY PARALLEL COMPUTING

- MPPA® processors can be tiled together, in package or on board
- MPPA® processors can easily be integrated into a system with other processors (FPGA, CPU, GPU)

Efficiency of CPUs, DSPs, FPGAs, ASICs (ISSCC)



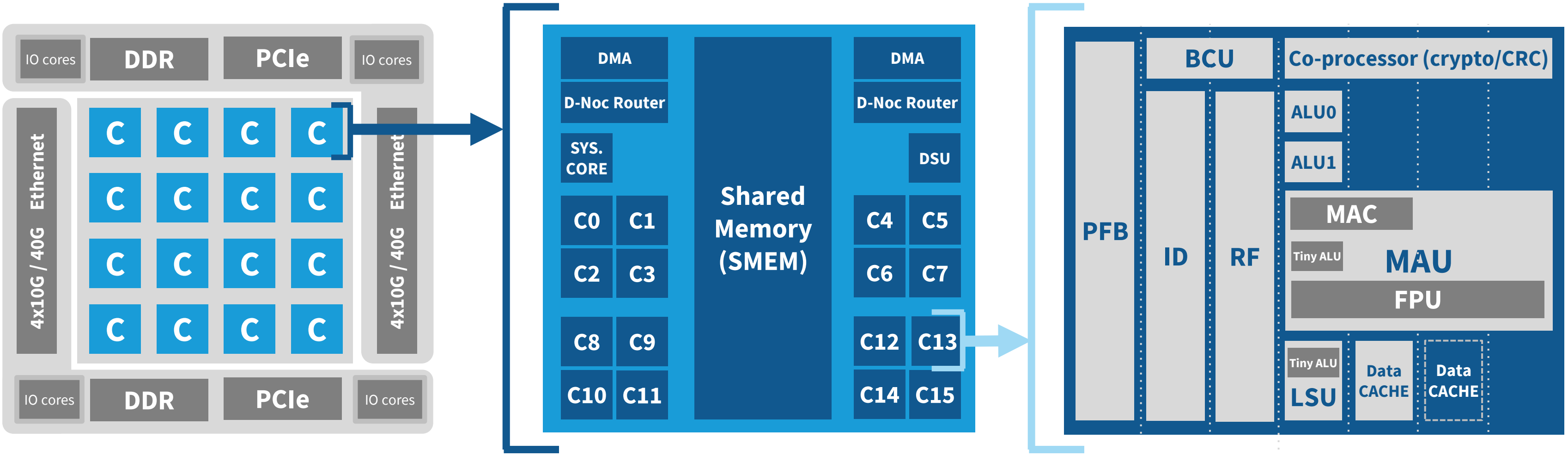
Source: ISSCC Proceedings

GPUs at same energy efficiency as DSPs

In Production

MPPA®-256 Bostan

TSMC CMOS 28HP, 600MHz



MANYCORE PROCESSOR

Architecture: Distributed memory

- 16 compute clusters
- 2 I/O clusters (2x quad-core each)
- Data & control networks-on-chip (NoC)

Performance

- 1 TFLOPS SP

Devices

- DDR3, 4 Ethernet 10G and 8 PCIe Gen3

COMPUTE CLUSTER

Architecture

- 16 user cores (SMP) + 1 system core

Communication

- NoC Tx and Rx interfaces

Memory:

- 2 MB multi-banked shared (77GB/s Shared Memory BW)

Debug

- Debug Support Unit (DSU)

VLIW CORE

Architecture

- 32-bit or 64-bit addresses
- 5-issue VLIW architecture
- MMU + I&D cache (8KB+8KB)
- 32-bit/64-bit IEEE 754-2008 FMA FPU

Security

- crypto co-processor (AES/SHA/CRC/…)

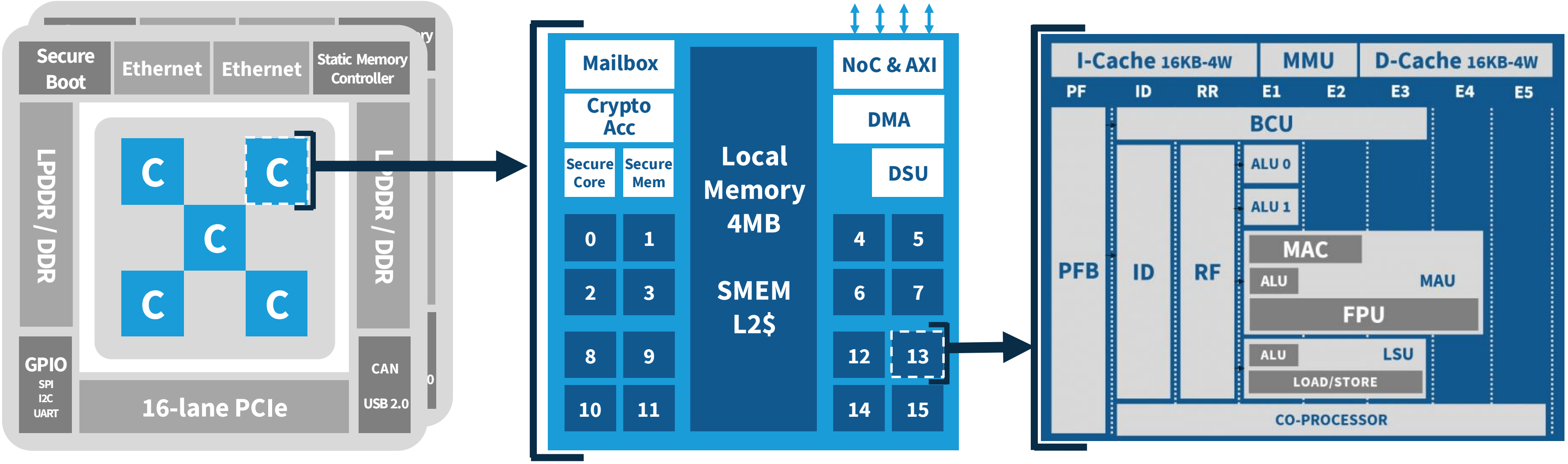
Performance

- 6 GFLOPS SP per core

Samples
Q1 2019

MPPA®-80 Coolidge

TSMC CMOS 16FFC, 1.2 GHz



MANYCORE PROCESSOR

Architecture updates

- 80 or 160 CPU cores
- 600/900/1200MHz frequency modes

Memory

- L2 cache coherency between clusters
- L2 refill in DDR and Direct access to DDR from clusters

COMPUTE CLUSTER

Architecture updates

- 16 CPU VLIW cores 64bits
- 16 Tensor co-processors
- Safety/Security dedicated core

Memory

- L1 cache coherency (configurable)
- 4MB memory configurable (614GB/s)

3RD GENERATION VLIW CORE

Architecture updates

- 64-bit core
- 6-issue VLIW architecture
- MMU + I&D cache (16KB+16KB)
- 16-bit/32-bit/64-bit IEEE 754-2008 FPU
- Vision/CNN tightly coupled co-processor

MPPA3® Coolidge NoC

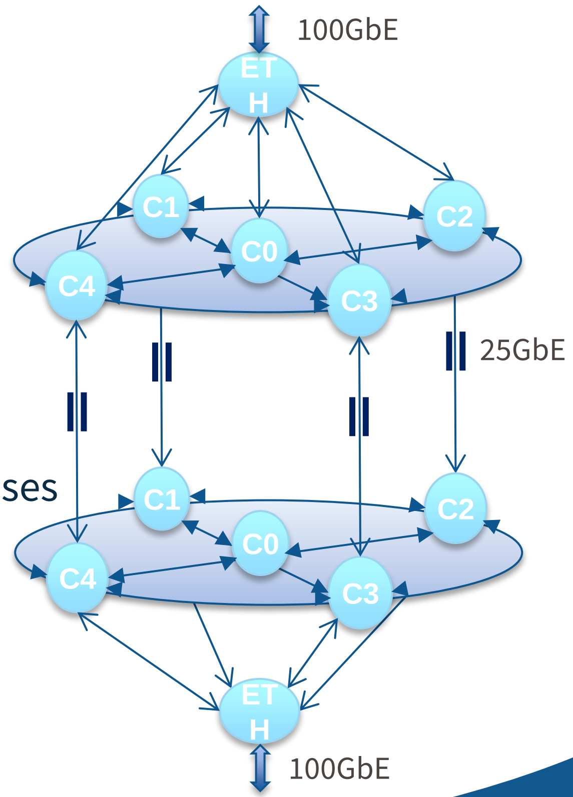
MPPA3® NoC architecture

- Wormhole switching with source routing
- 2 virtual channels, 4x TX DMA channels
- RDMA, remote queues, remote atomics
- 128-bit flits, up to 17 flits/packet (256B payload)

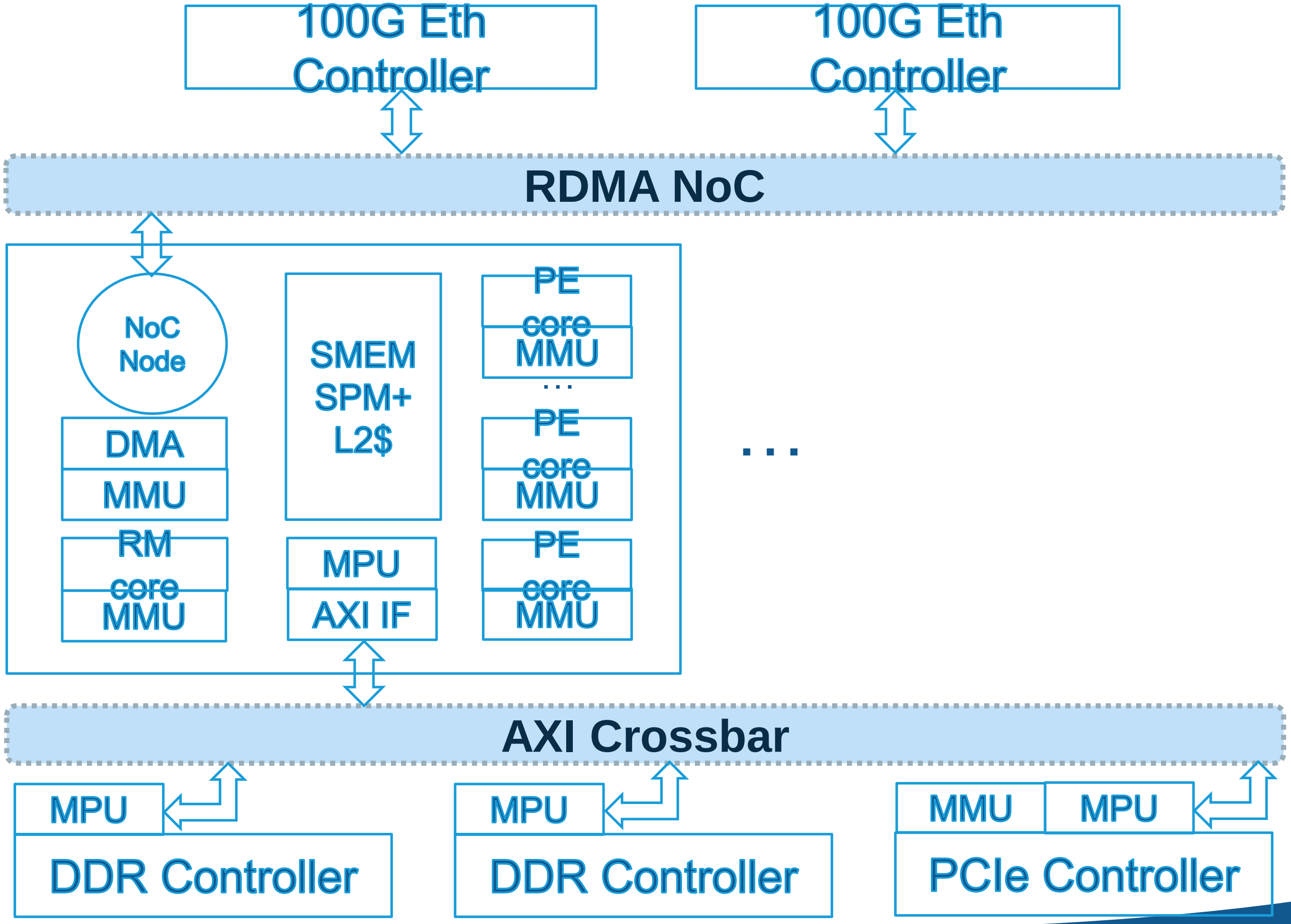
4x 25Gbps Ethernet lanes reused for NoC extension

- NoC packet encapsulation into IEEE 802.1Q standard for VLAN
- Designed for direct connections between 2 to 4 chips (using FEC)
- VCs map to IEEE 802.1Qbb Priority-based Flow Control (PFC) classes

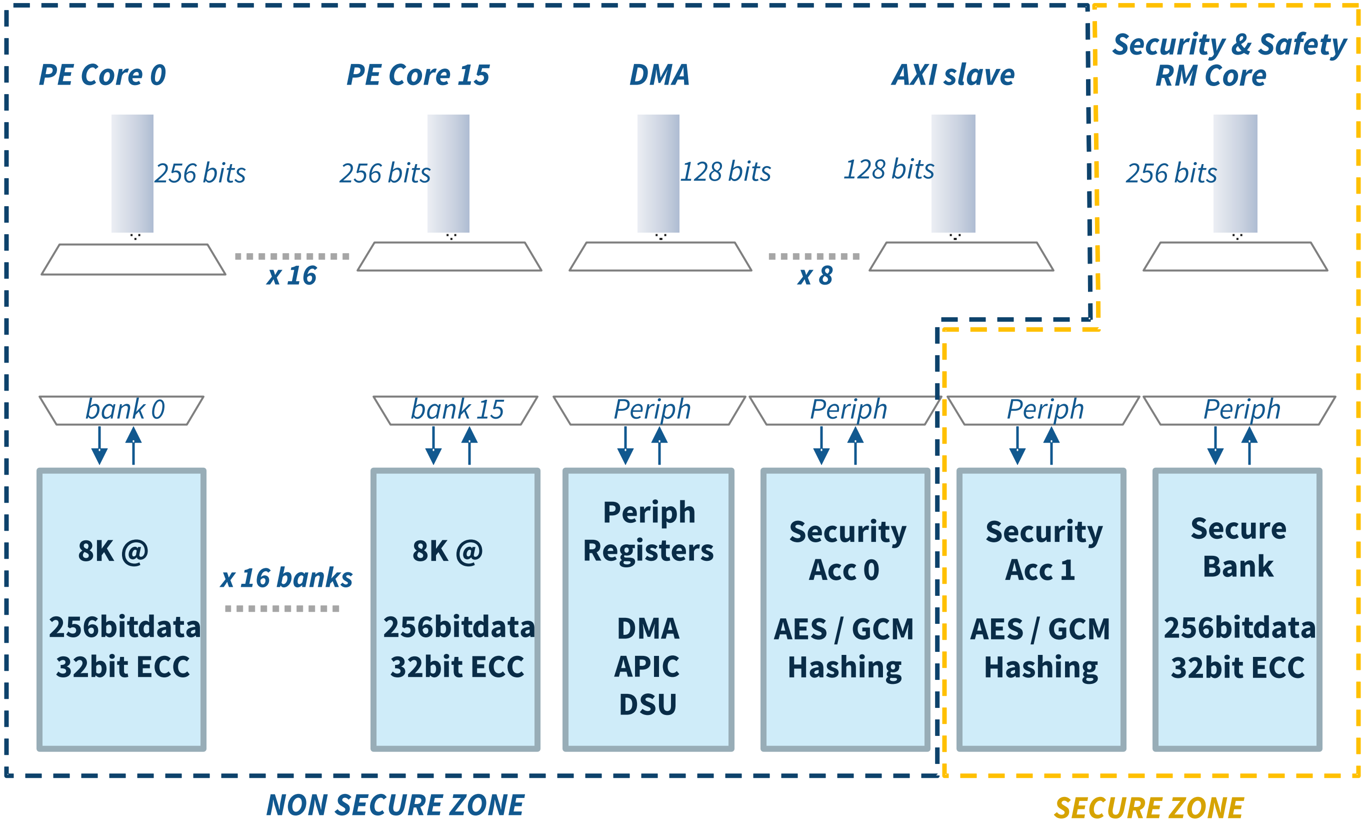
MAC dst 6 bytes	MAC src 6 bytes	VLAN etype 0x8100 2 bytes	VLAN TCI PFC (3 bits) / CFI (1 bit) NoC pkt nb (12 bits) 2 bytes	NoCX etype 0xB000 2 bytes	NoC pkt0	NoC pkt1	FCS 4 bytes



MPPA3® Coolidge Global Interconnects



MPPA3® Coolidge Compute Cluster


PE Core 0
PE Core 15
DMA
AXI slave
Security & Safety RM Core
256 bits
256 bits
128 bits
128 bits
256 bits
x 16
x 8
bank 0
bank 15
Periph
Periph
Periph
Periph
8K @ 256bitdata 32bit ECC
x 16 banks
8K @ 256bitdata 32bit ECC
Periph Registers DMA APIC DSU
Security Acc 0 AES / GCM Hashing
Security Acc 1 AES / GCM Hashing
Secure Bank 256bitdata 32bit ECC
NON SECURE ZONE
SECURE ZONE

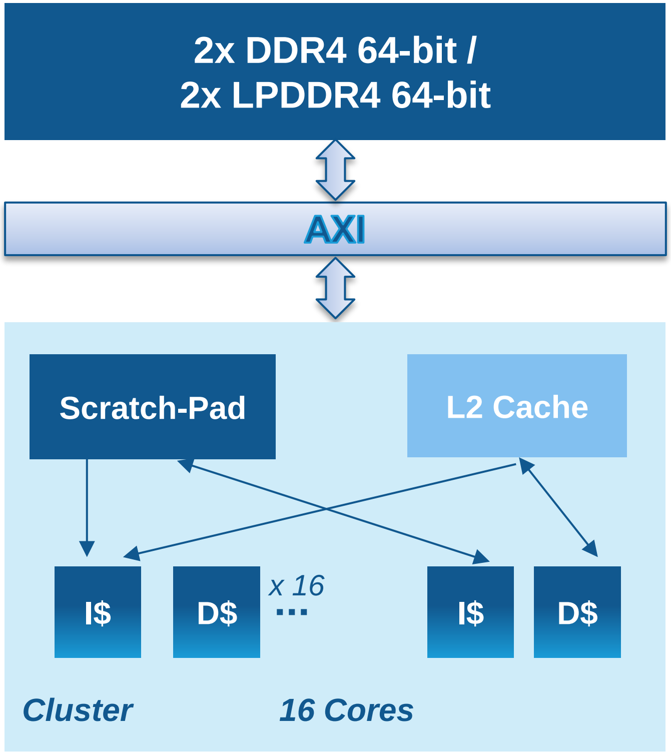
MPPA3® Coolidge Memory Hierarchy

VLIW Core L1 Caches

- 16KB / 4-way LRU instruction cache per core
- 16KB / 4-way LRU data cache per core
- 64B cache line size
- Write-through, write no-allocate (write around)
- Coherency configurable across all L1 data caches
- DMA writes are L1 cache-coherent

Cluster L2 Cache & Scratch-Pad Memory

- Scratch-pad from 2MB to 4MB
  - 16 independent banks, full crossbar
  - Interleaved or banked address mapping
- L2 cache from 0MB to 2MB
  - 16-way Set Associative
  - 256B cache line size
  - Write-back, write allocate
  - Optionally coherency across clusters



L1 cache coherency	L2 cache coherency
enable /disable	enable /disable

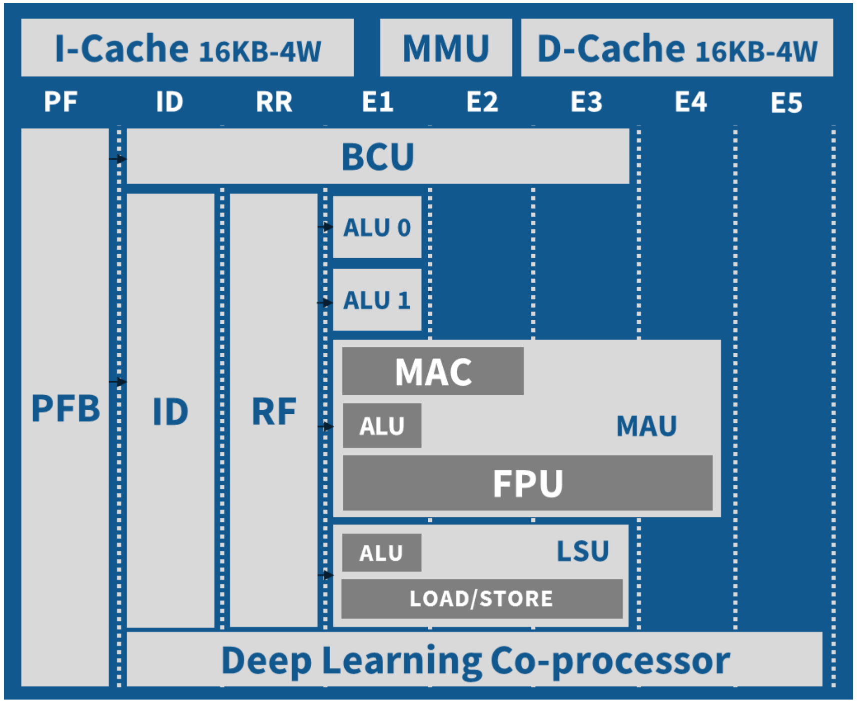
MPPA3® Coolidge 3rd Generation 64-Bit VLIW Core

Unified scalar and SIMD ISA

- 64x 64-bit general-purpose registers
- Operands can be single registers, register pairs (128-bit) or register quadruples (256-bit)
- Immediate operands up to 64-bit, including F.P.
- 128-bit SIMD instructions by dual-issuing 64-bit on the two ALUS or by using the FPU datapath

FPU capabilities

- 64-bit x 64-bit + 128-bit → 128-bit
- 128-bit op 128-bit → 128-bit
- FP16x4 SIMD 16 x 16 + 32 → 32
- FP32x2 FMA, FP32x4 FADD, FP32 FMUL Complex
- FP32 Matrix Multiply 2x2 Accumulate



COOLIDGE VLIW CORE PIPELINE

MPPA3® Coolidge Tensor Coprocessor

Extend core ISA with « wide » SIMD

- 64x 256-bit wide vector register file
- Matrix-oriented arithmetic operations

Full integration into core pipeline

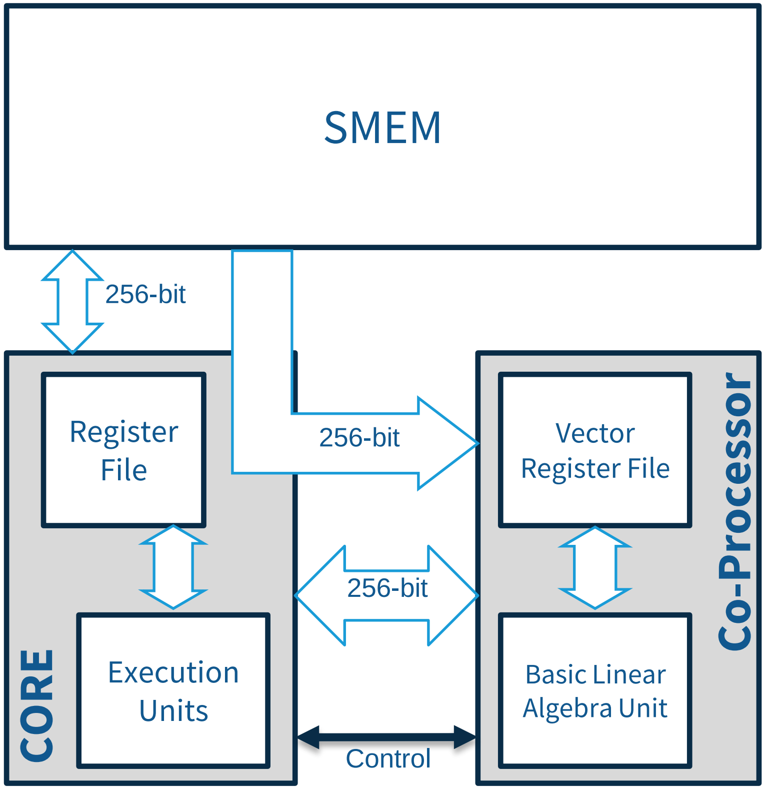
- Move instructions with matrix-transpose
- Proper dependency / cancel management

Leverage MPPA memory hierarchy

- SMEM directly accessible from coprocessor
- Memory load stream aligment operations

Arithmetic performances

- 128x INT8→INT32 MAC/cycle
- 64x INT16→INT64 MAC/cycle
- 16x FP16→FP32 FMA/cycle

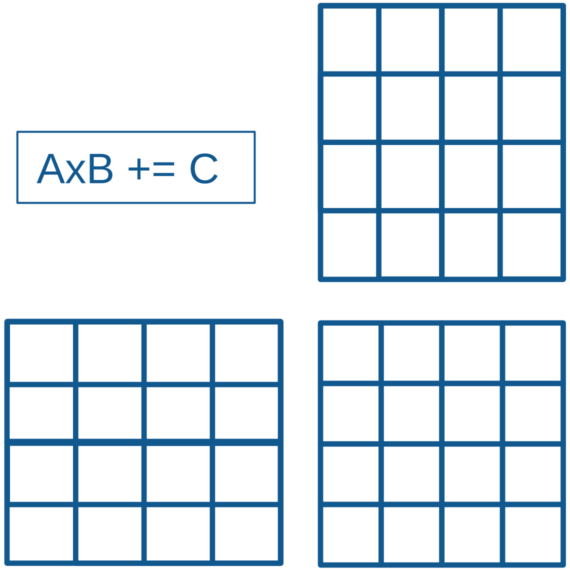


MPPA3® Tensor Coprocessor Matrix Operations

- INT16 to INT64 convolutions:

(4x4)int16 . (4x4)int16 += (4x4)int64

16x DP4-ADD → 64 MAC/cycle

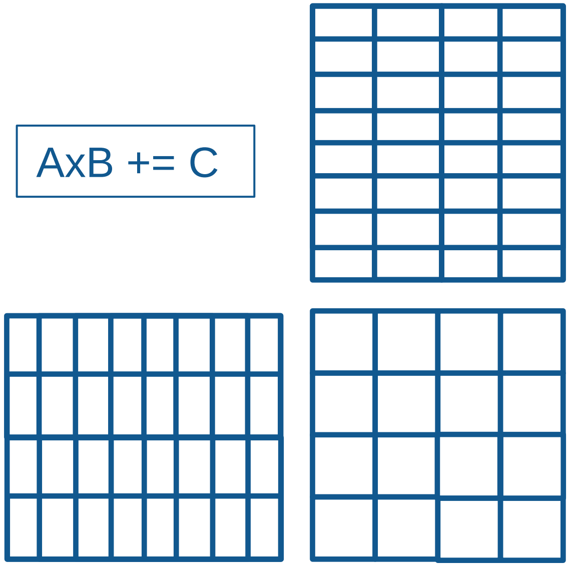


AxB += C

- INT8 to INT32 convolutions

(4x8)int8 . (8x4)int8 += (4x4)int32

16x DP8-ADD → 128 MAC/cycle



AxB += C

KONIC80, MPPA®-256 Bostan PCIe Board

Features

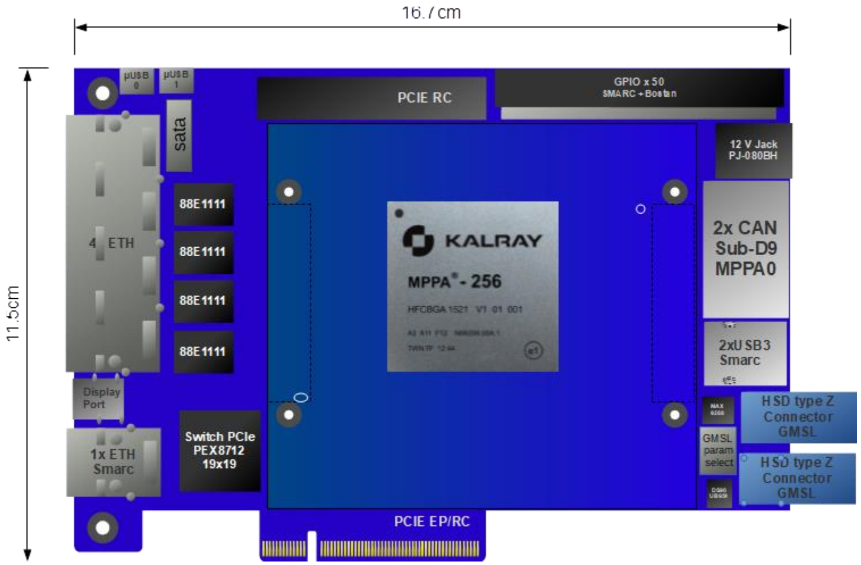
- 80GbE (2x40GbE, 8x10GbE) full duplex, line rate
- PCIe Gen3 16-lanes providing a throughput of up to 128Gbps Full duplex
- 2,500 instructions per packet @240Mpps
- 256 C/C++ programmable cores
- 1 TOPS
- Low-power/20W typical
- 40MB on-chip memory + 5MB caches
- 2x 4GB DDR3
- Dedicated HW for packet acquisition, classification and emission
- True Random Number Generator (TRNG)
- 128 Crypto Co-Processors for 80Gbps full-duplex MACSec, IPSec and SSL offload

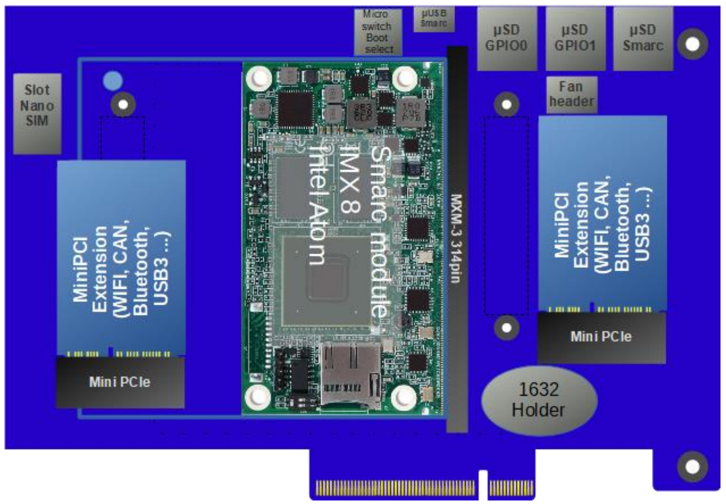
- Software
  - OpenDataPlane SDK
  - Virtualization Offload
    - VXLAN, NVGRE, GENEVE, TRILL
    - OVS offload
  - Storage virtualization
    - iSCSI termination
    - virtio storage interfaces
  - Kernel-bypass
    - DPDK, ODP, socket

AB06 Board for MPPA® Bostan and MPPA® Coolidge

The MPPA® processor is mounted on a SOM (System on Module) mezzanine board with DDR, PCIe, Ethernet, CAN

An optional host CPU SOM (x86, ARMv8) can be plugged on the other side





Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

AccessCore™ Software Development Kit

OPEN
SOFTWARE
&
TOOLS

Cohesive Coding Environment

- All cores implement the same Instruction Set Architecture (ISA)

Open Standards Programming

- Supports C, C++, OpenMP 3, OpenCL 1.2 programming models

Software Development Tools

- Eclipse, GCC, GDB, LLVM, Trace, etc.

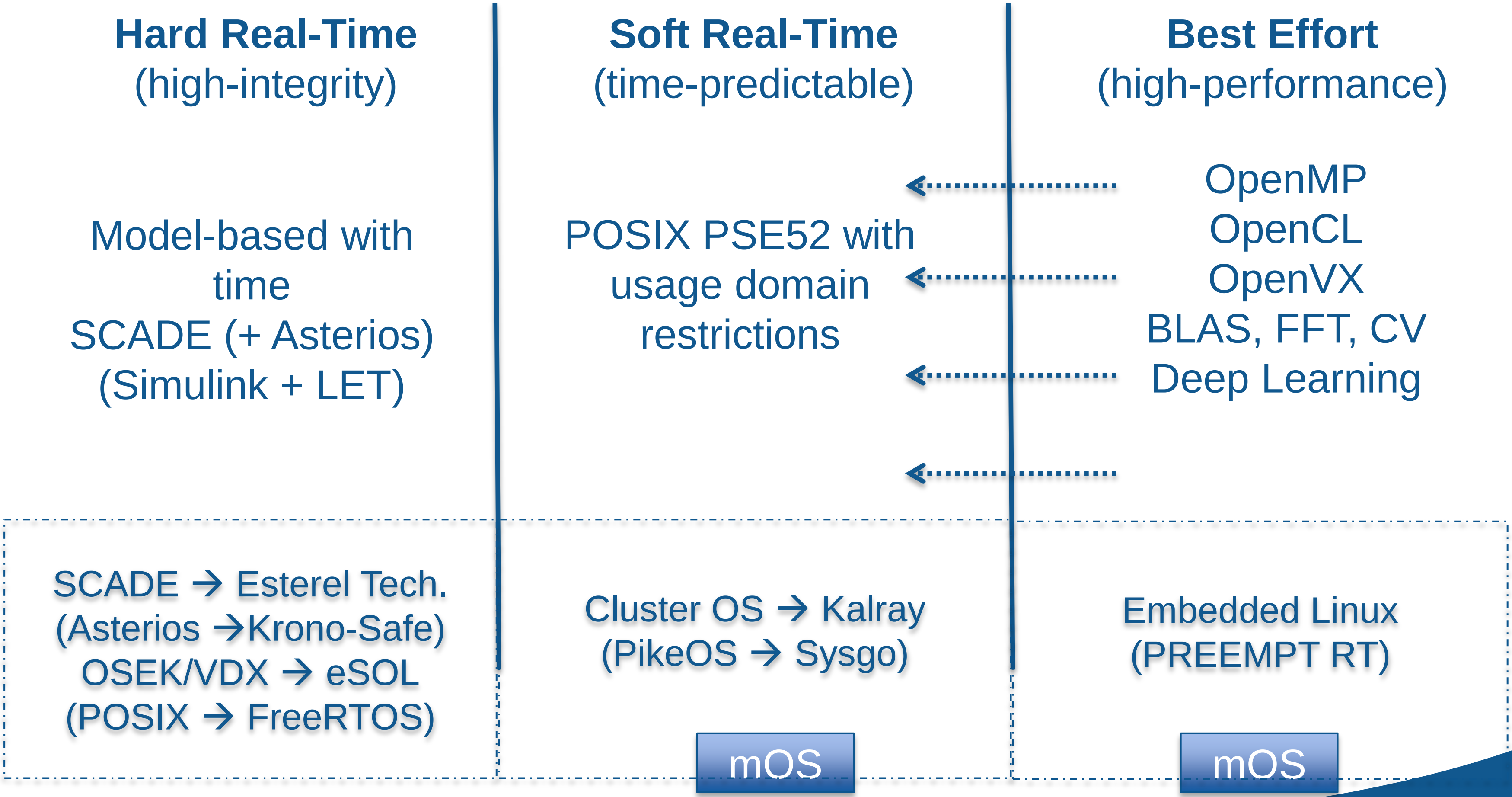
Operating Systems

- Linux kernel and I/O drivers on I/O clusters
- Lightweight POSIX OS on compute clusters

Tools & Libraries

- Code generator for deep learning inference
- Optimized BLAS and FFT libraries

MPPA® Embedded Platform Roadmap



Hard Real-Time
(high-integrity)

Model-based with time
SCADE (+ Asterios)
(Simulink + LET)

SCADE → Esterel Tech.
(Asterios →Krono-Safe)
OSEK/VDX → eSOL
(POSIX → FreeRTOS)

Soft Real-Time
(time-predictable)

POSIX PSE52 with usage domain restrictions

Cluster OS → Kalray
(PikeOS → Sysgo)

mOS

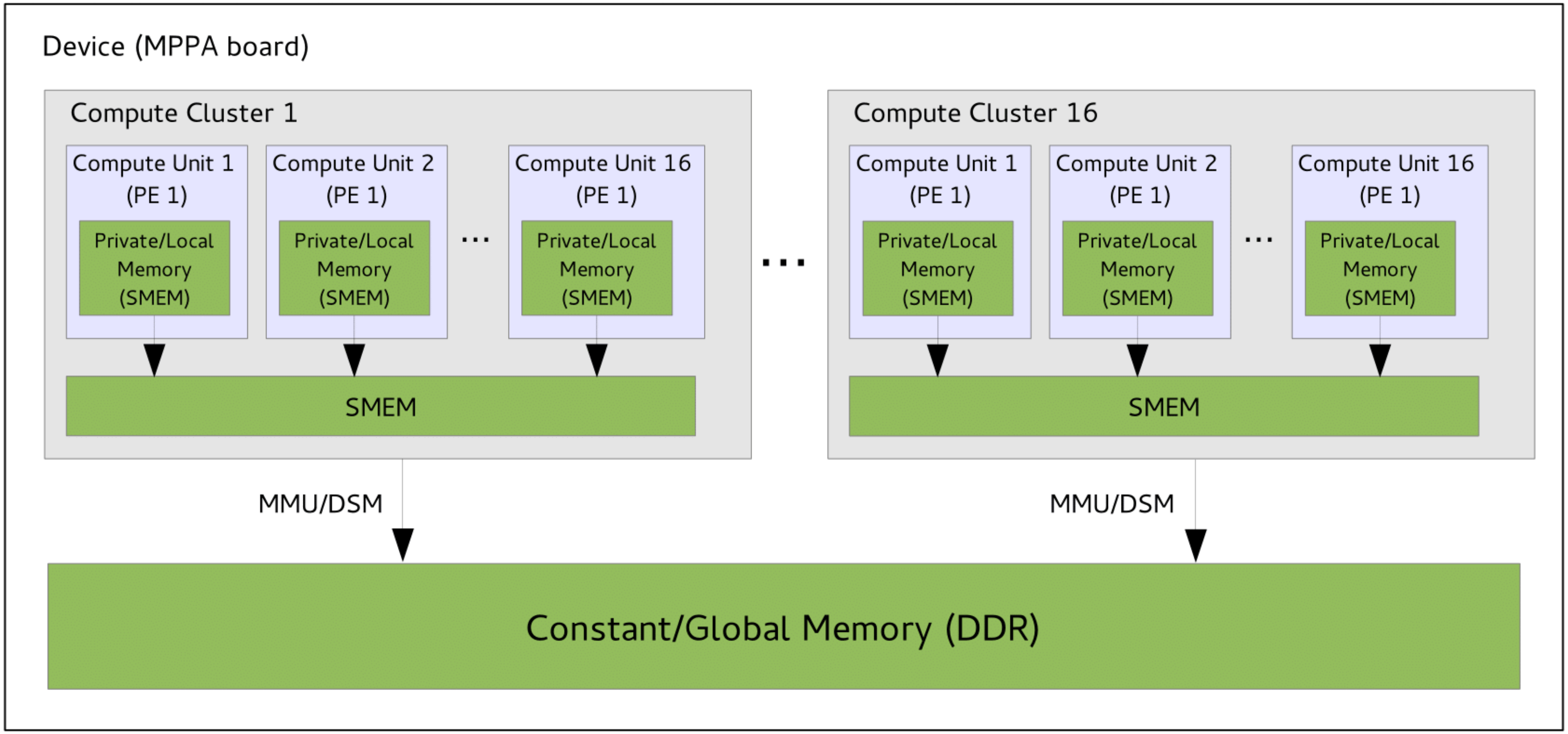
Best Effort
(high-performance)

OpenMP
OpenCL
OpenVX
BLAS, FFT, CV
Deep Learning

Embedded Linux
(PREEMPT RT)

mOS

OpenCL 1.2 on the MPPA Platform



OpenCL Issues on a CPU-Based Manycore Processor

Difference between GPGPUs and manycore processors based on CPUs or DSPs

- No core hardware multithreading for automatic overlapping of memory latencies
- Significant benefits from direct communication between Work Groups (non-standard)
  - Avoid using the external memory (Global Memory) for data transfers

From TI KeyStone ‘Optimization Techniques for Device (DSP) Code’

- Prefer Kernels with 1 work-item per work-group (DSP seen as one Compute Unit)
- Use async_work_group_copy and async_work_group_strided_copy
  - “it is almost always better to write the values to a local buffer and then copy that local buffer back to a global buffer using the OpenCL async_work_group_copy function”

On the MPPA, extend the standard OpenCL asynchronous copies

- OpenCL asynchronous copies are restricted to dense local memory accesses
- Need to provide enough local memory => 1 Work Group per Cluster preferred
- Extensions for 2D/3D accesses in global memory (done on ST P2012 OpenCL)

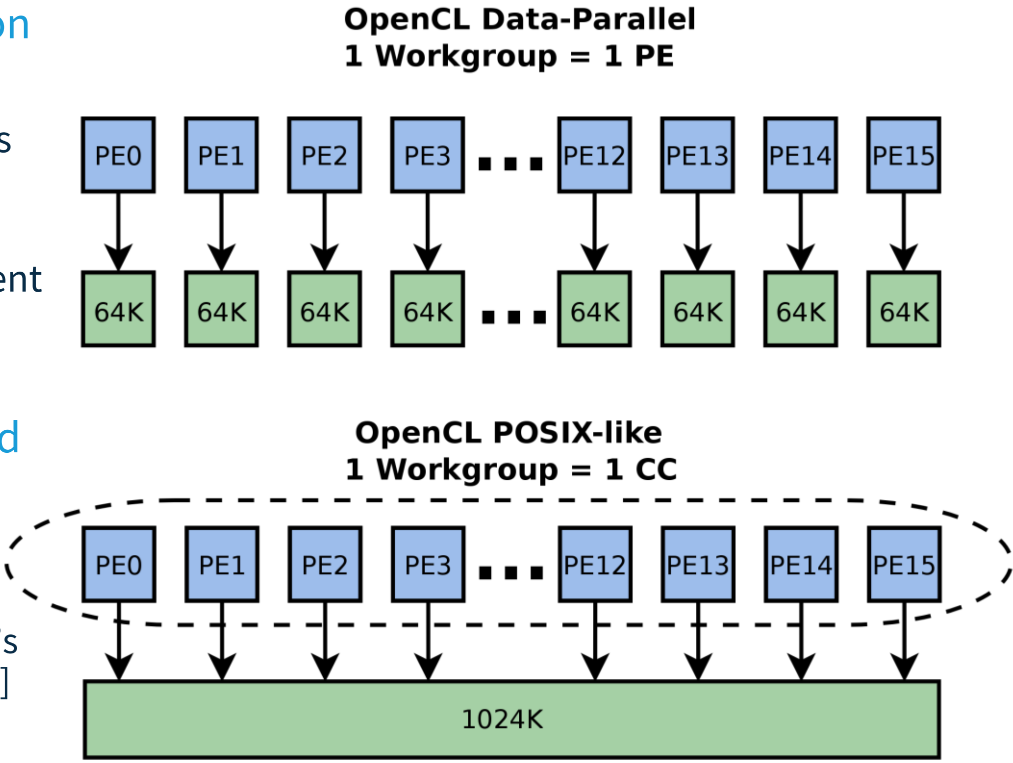
OpenCL Data-Parallel and Task-Parallel+POSIX

Parts of standard OpenCL that are useful on a CPU-based manycore processor

- Host program allocates global buffers, creates executable kernels, and dispatches work in queues
- Kernel invocation with a user-defined argument list, which distinguishes between local and global objects

OpenCL extensions required for CPU-based manycore processors

- Kernel code in standard C/C++/OpenMP and/or assembly language
- Kernel code with classic CPU multi-threading [TI's "OpenMP Dispatch With OpenCL" on KeyStone-II]
- Kernel code that accesses the local memory of other Compute Units



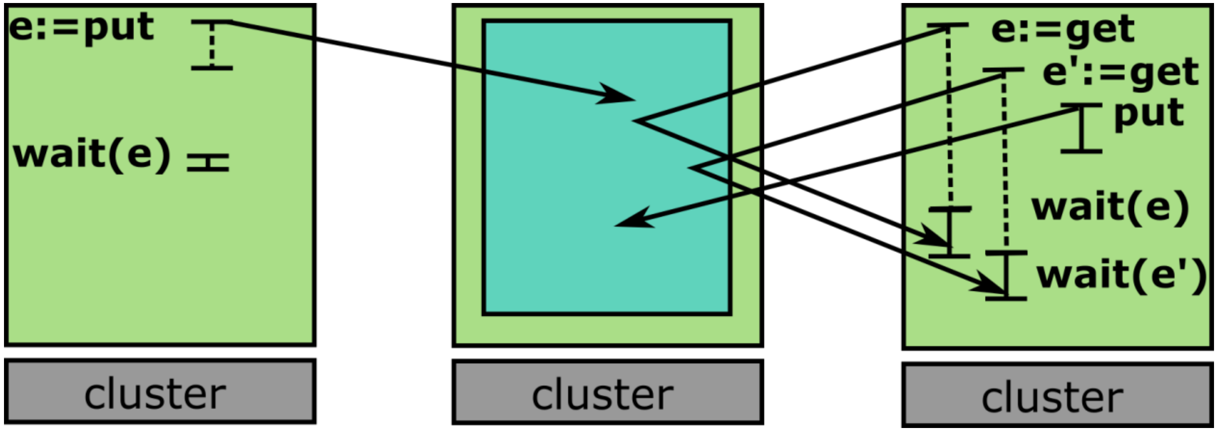
MPPA Asynchronous Operations Principles (1)

Inspired by HPC clusters one-sided communication & synchronization

- Cray SHMEM, ORNL ARMCI, Berkeley GasNet, MPI-3 one-sided subset
- Cannot directly reuse these libraries because of the MPPA architecture

Asynchronous remote data transfers

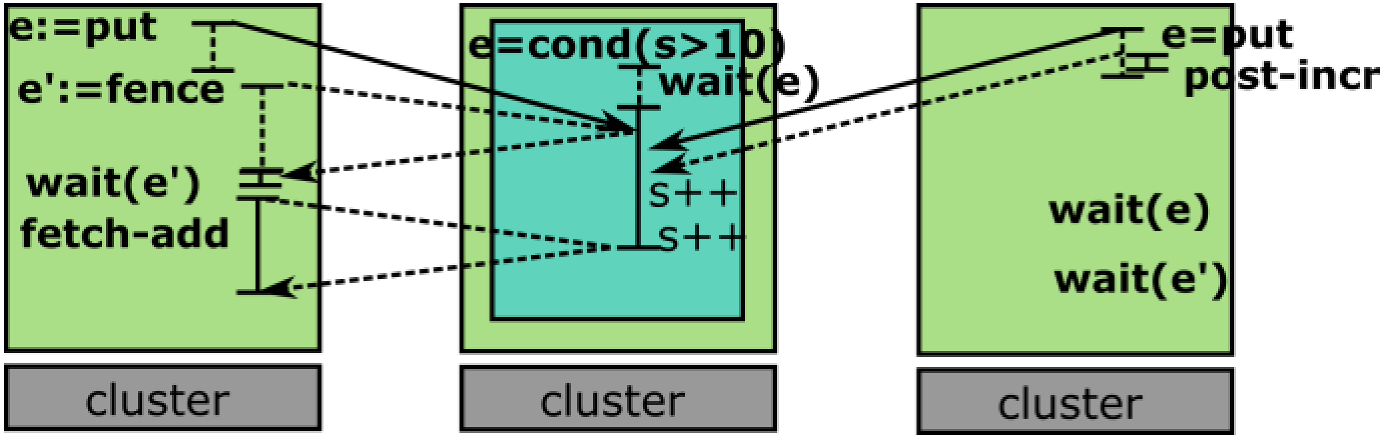
- Put (remote write) and Get (remote read) operations with data reshaping
- All data transfer operations return immediadely to caller
- An event structure can be used to wait/test for local completion



MPPA Asynchronous Operations Principles (3)

Point-to-point synchronization operations

- Remote fence (global completion), peek, poke, post-add, fetch-clear, fetch-add
- Locally, wait for the comparison between a local variable and a value to be true
- No busy waiting, only lock-free data structures



Remote queues N to 1 (Rqueues)

- Push on a remote queue-like memory segment, with atomicity if possible
- Classic distributed synchronization primitive, foundation of active messages

MPPA Asynchronous Operations API Overview

Dense Transfers

- mppa_async_get
- mppa_async_put
- mppa_async_get_spaced
- mppa_async_put_spaced
- mppa_async_get_indexed
- mppa_async_put_indexed

Sparse Transfers

- mppa_async_sget_spaced
- mppa_async_sput_spaced
- mppa_async_sget_blocked2d
- mppa_async_sput_blocked2d
- mppa_async_sget_blocked3d
- mppa_async_sput_blocked3d

Asynchronous Events

- mppa_async_event_wait
- mppa_async_event_test

Global Synchronization

- mppa_async_fence
- mppa_async_peek
- mppa_async_poke
- mppa_async_postadd
- mppa_async_fetchclear
- mppa_async_fetchadd
- mppa_async_evalcond

Remote queues

- mppa_async_enqueue
- mppa_async_dequeue

Illustration of Code Transformations for Put/Get

Example extracted from a tiled matrix multiply algorithm

- Inner loop is first converted to a dense Put (mppa_async_put)
- Outer loop is then converted to a sparse Put (mppa_async_sput_spaced)
- Use of blocking calls (last Put parameter is NULL instead of event pointer)

```
void
tileto(int m, int n, dtype C[m][n], int i, int j, int p, dtype c[p][p])
{
  int mii = MIN(p, m-i);
  int mjj = MIN(p, n-j);
#ifndef useputget
  for (int ii = 0; ii < mii; ii++) {
    for (int jj = 0; jj < mjj; jj++) {
      C[i+ii][j+jj] = c[ii][jj];
    }
  }
#elif (useputget == 1)
  for (int ii = 0; ii < mii; ii++) {
    mppa_async_put(&c[ii][0], &C[i+ii][j+0], ddr0_segment, mjj*sizeof(dtype), NULL);
  }
#elif (useputget == 2)
  mppa_async_sput_spaced(&c[0][0], &C[i+0][j+0], ddr0_segment, mjj*sizeof(dtype), mii,
                         (char*)&c[1][0]-(char*)&c[0][0], (char*)&C[i+1][j+0]-(char*)&C[i+0][j+0], NULL);
#endif//useputget
}
```

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

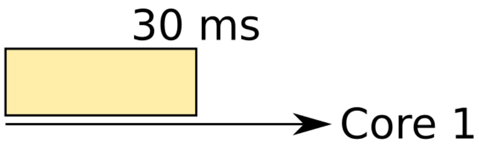
SCADE Code Generation for the MPPA®

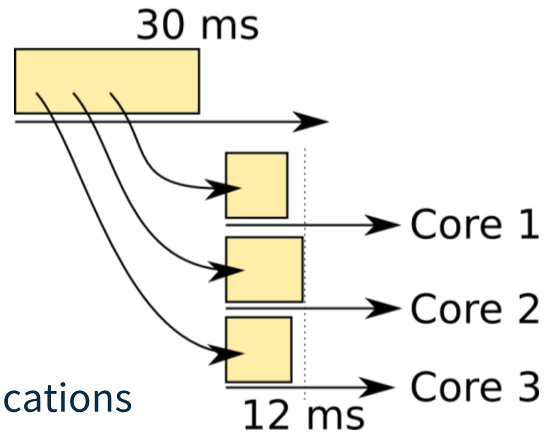
Safety-critical control-command applications

- Model-based programming using SCADE Suite® from Esterel Technologies
- Complemented with static timing analysis of binary code (aiT from AbsInt)
- Retargeting of the formally proven bug-free CompCert C99 compiler

Motivations for multicore and manycore execution

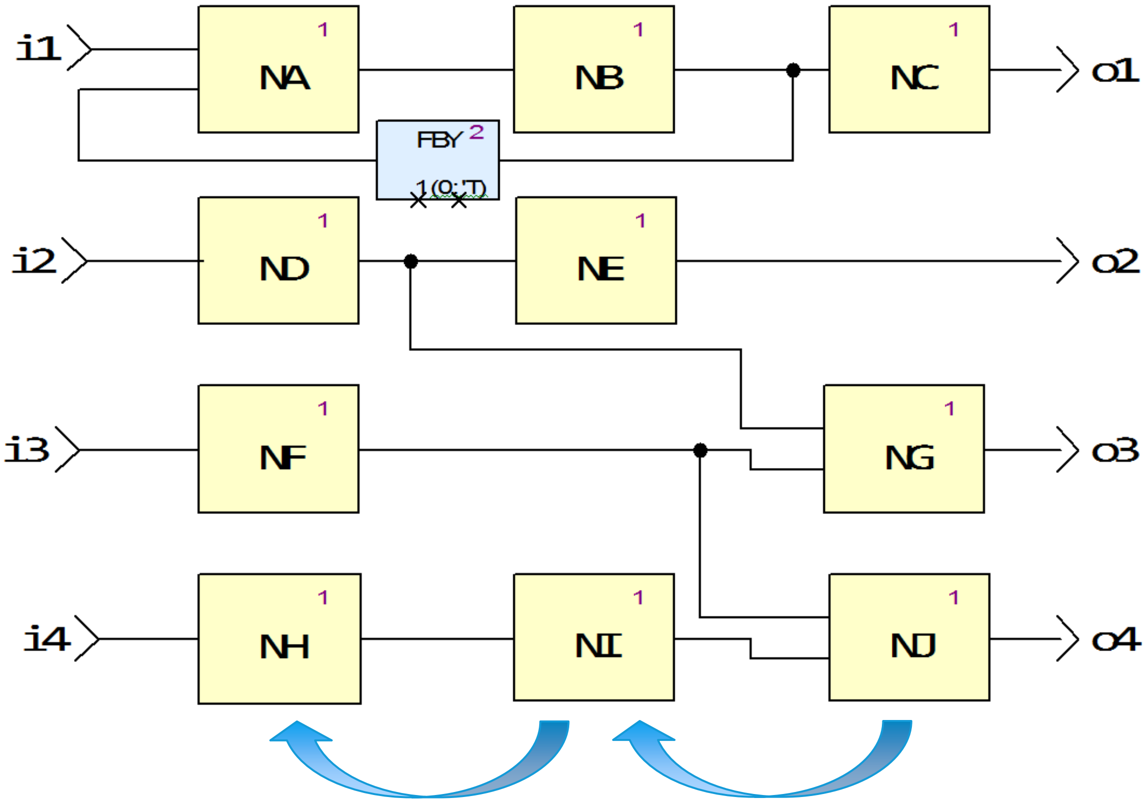
- Distribute the compute load across cores and reduce memory interferences





- Effective implementation of multi-rate harmonic applications
- Envision use of fast Model Predictive Control (MPC) techniques

Example of SCADE Suite Program



Dependencies represented by wires.

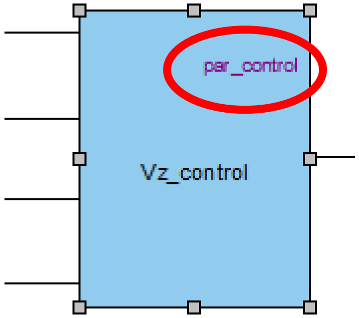
SCADE Suite Program Input Model

Group several operator instances in “parallel subsets”

- Parallel subsets can be nested
- Compiler verifies that parallel subset are dependence-free
- Instances of the same subset can be in different operators (if they end up in the same unexpanded operator)
- Each instance in a subset may be executed on a different “thread”

Partitioning is captured using user annotations

- Scade model is unchanged
- Occurrence pragma beginning by the prefix “#par_”
- The suffix is the identifier of the parallel subset
- Textual & graphical format



```
o1, o2 = #par_SetName MyNode(i1, i2, i3);
```

KCG OpenMP Code Generation

Rely on OpenMP 2.5 features

- One parallel region for each parallel subset
- Task parallelism (omp section) for operators
- Data parallelism (omp for) for iterators

Dynamic thread scheduling

- The OpenMP runtime is provided by the C/C++ compiler (GCC)

```
function imported N1(i:int32) returns (o:int32);
function imported N2(i:int32) returns (o:int32);
function imported N3(idx : int32) returns (o:int32);

function root(i1,i2:int32) returns (z:int32)
var x,y:int32; a:int32^10;
let
  x = #par_1 N1(i1);
  y = #par_1 N2(i2);
  a = (#par_1 mapi N3 <<10>>)();
  z = x + y + a[0];
tel
```

```
void root(inC_root *inC,
          outC_root *outC)
{
 array_int32_10 a;
 kcg_size idx;
 kcg_int32 x,y;

 /* par_1 */
 #pragma omp parallel
 {
   #pragma omp sections nowait
   {
     #pragma omp section
     x = N1(inC->i1);

     #pragma omp section
     y = N2(inC->i2);
   }

   #pragma omp for nowait
   for (idx = 0; idx < 10; idx++) {
     a[idx] = N3((kcg_int32) idx);
   }
 }
 outC->z = x + y + a[0];
}
```

KCG Task-Based Code Generation

Overview

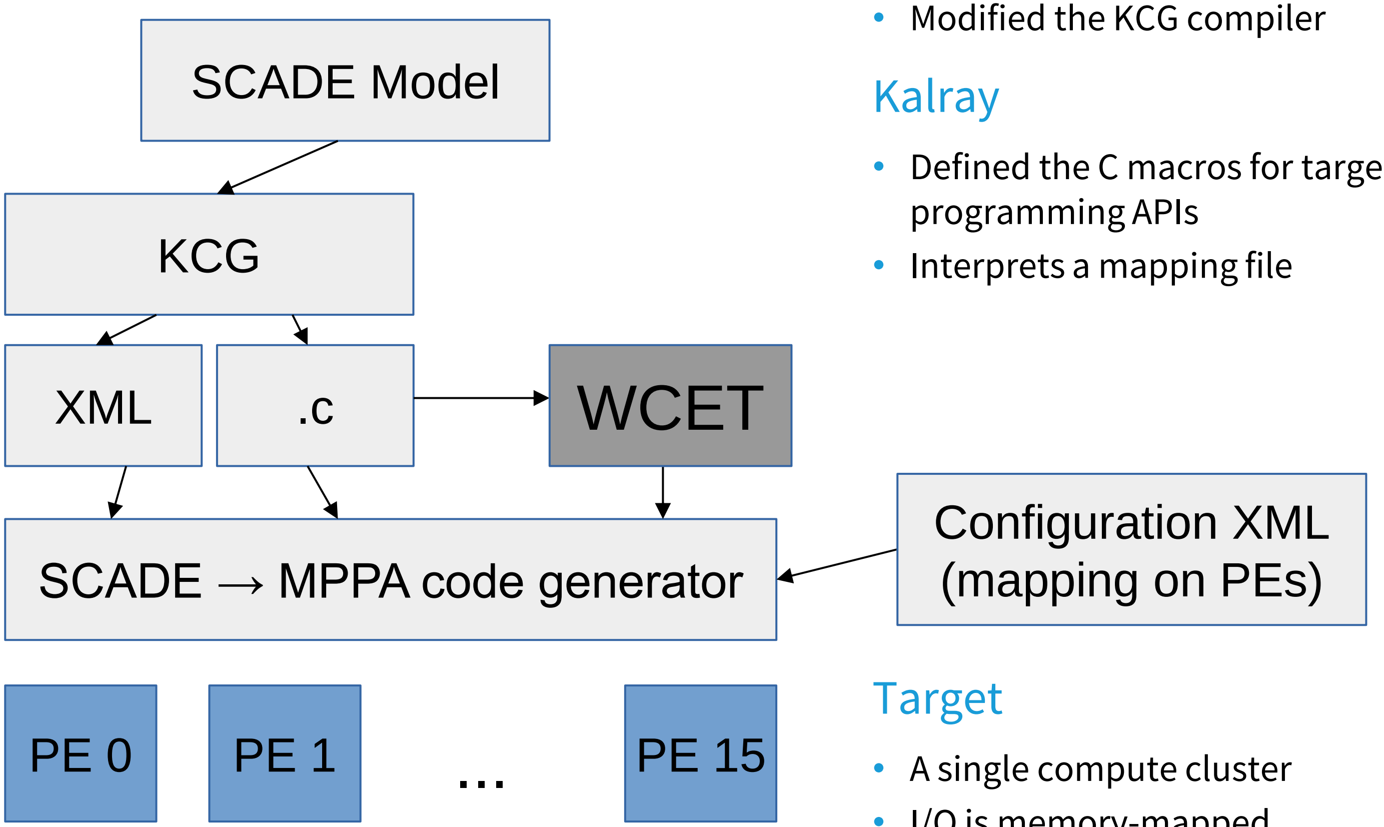
- Generate tasks that communicate with one-to-one channels (KPN)
  - One task for the root operator
  - One task for each instance of operator in a parallel subset
- Developed in the setting of the ITEA3 ASSUME project
  - Also support AbsInt aiT and INRIA CompCert targeting to MPPA® cores

Properties

- Target agnostic: KCG uses macros for all target operations
- Instantiated for the Pthread and the MPPA Low-Level 'bare' runtime
- Code generation is independent from the allocation of tasks

```
void N_worker()
{
  recv(in_channel, i);  // receive inputs
  o = N(i);             // call operator
  send(out_channel, o); // send outputs
}
```

SCADE Workflow for the MPPA® Bostan Processor


SCADE Model
KCG
XML
.c
WCET
SCADE → MPPA code generator
Configuration XML (mapping on PEs)
PE 0
PE 1
…
PE 15

ANSYS

- Modified the KCG compiler

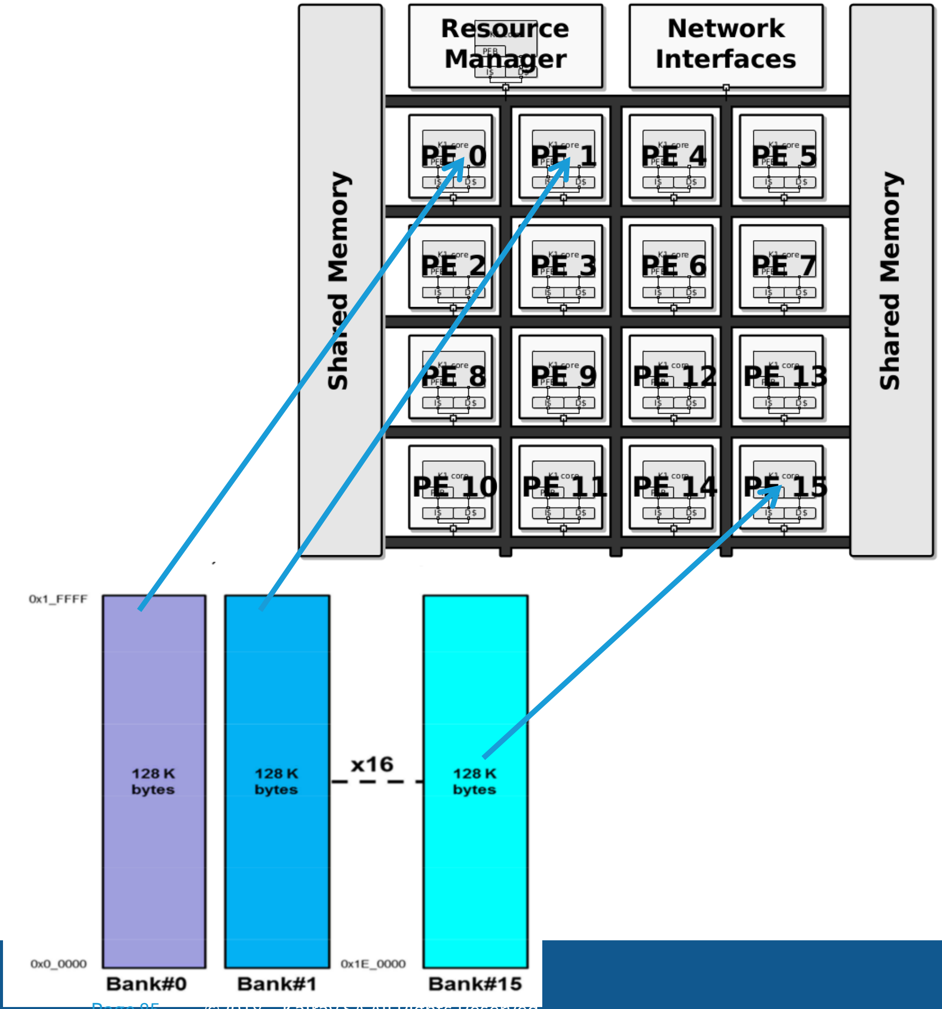
Kalray

- Defined the C macros for targeting the Low Level programming APIs
- Interprets a mapping file

Target

- A single compute cluster
- I/O is memory-mapped

Managing Local Memory Interference

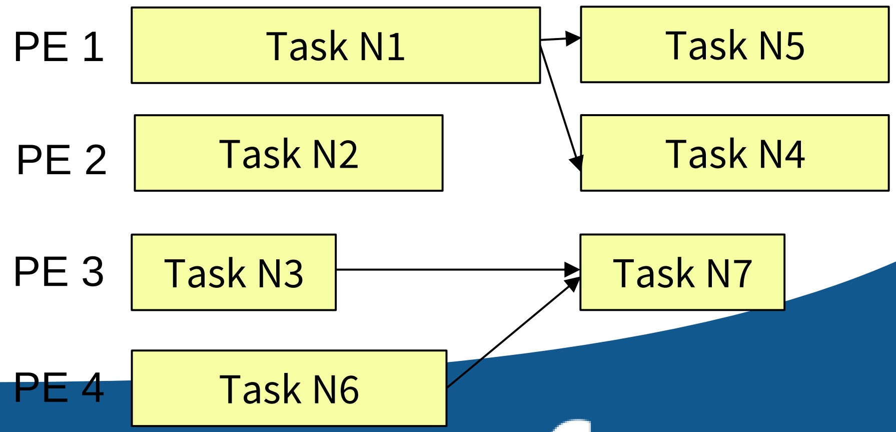


One memory bank per PE core

- Determined by a linker map and section attributes in code/data
- Non-interfering memory accesses except for channels

Communication interference

- Remote write policy for channel data: multicast to successors



Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

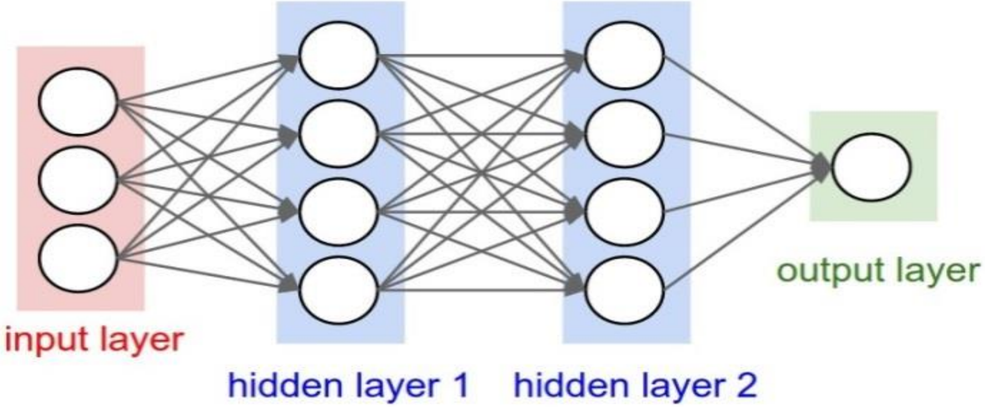
Deep Learning Inference

Conclusions

Artificial Intelligence

The science and engineering of creating intelligent machines. (John McCarthy, 1956)

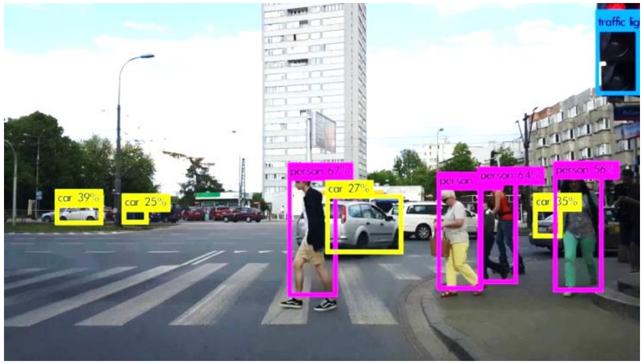
- **Machine Learning (ML):** Field of study that gives computers the ability to learn without being explicitly programmed (Arthur Samuel, 1959)
  - **Deep Learning (DL):** Allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction (Yann Le Cun et al., 2015)
    - **Convolutional Neural Networks (CNN):** Most filtering operations performed by feature maps are discrete convolutions



Machine Learning Steps

Training: Learning part– Off-line – Millions of data (images, sounds, …) – FP32

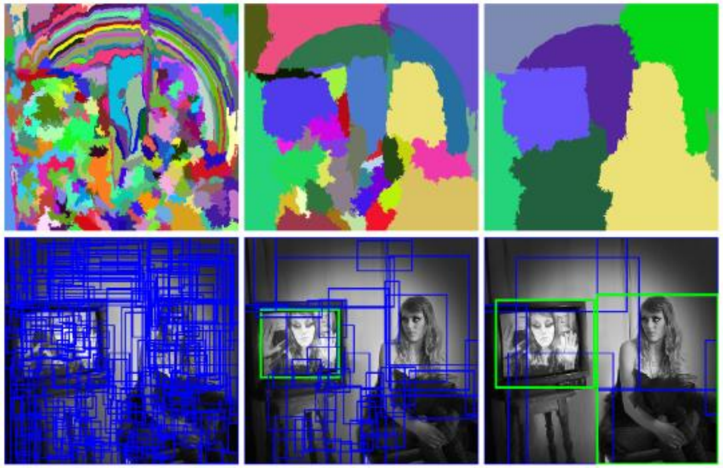
Inference: Classification / Recognition / Detection– On-line / Real time – FP16 / INT8

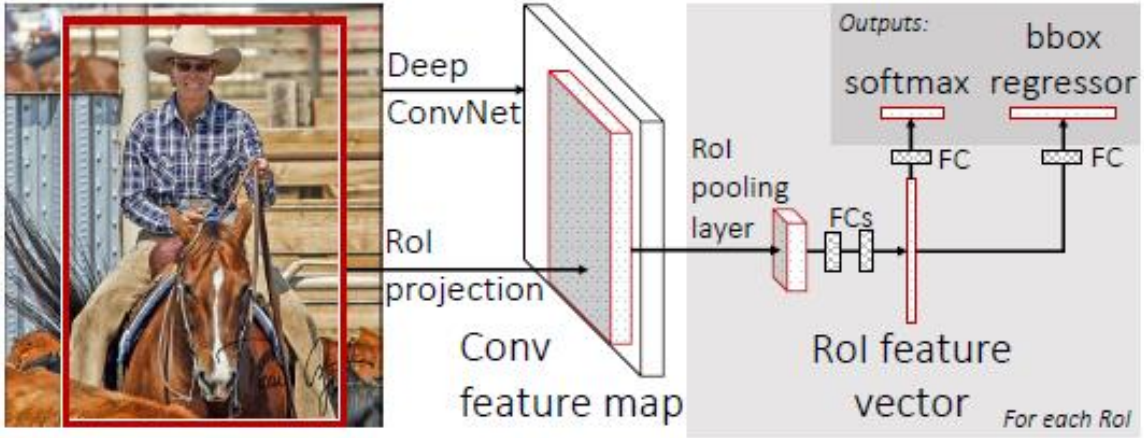


R-CNN, Fast & Faster R-CNN (Girshick & Ren, 2014-2016)

Regional CNN and improvements use two steps for object detection

1) *Proposal of candidate regions (initially by sementation, then by neural computing)*
2) *Classification of candidate regions (neural computing and refinment steps)*

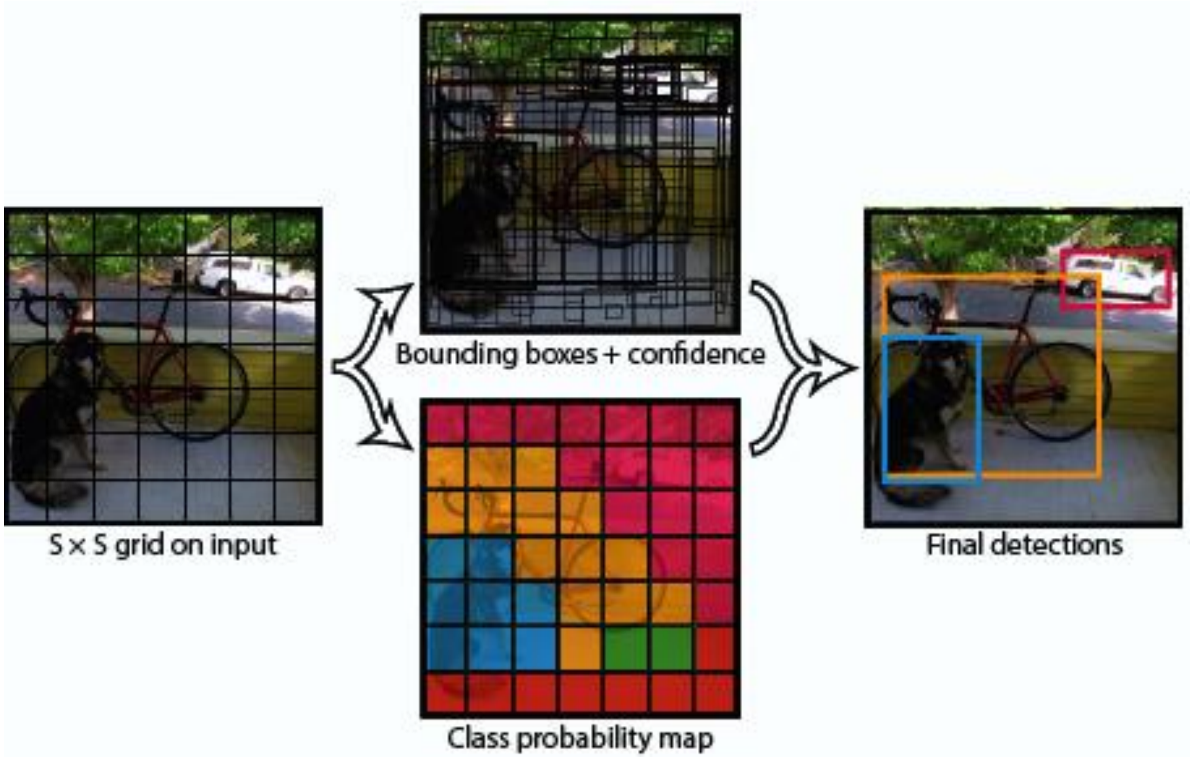




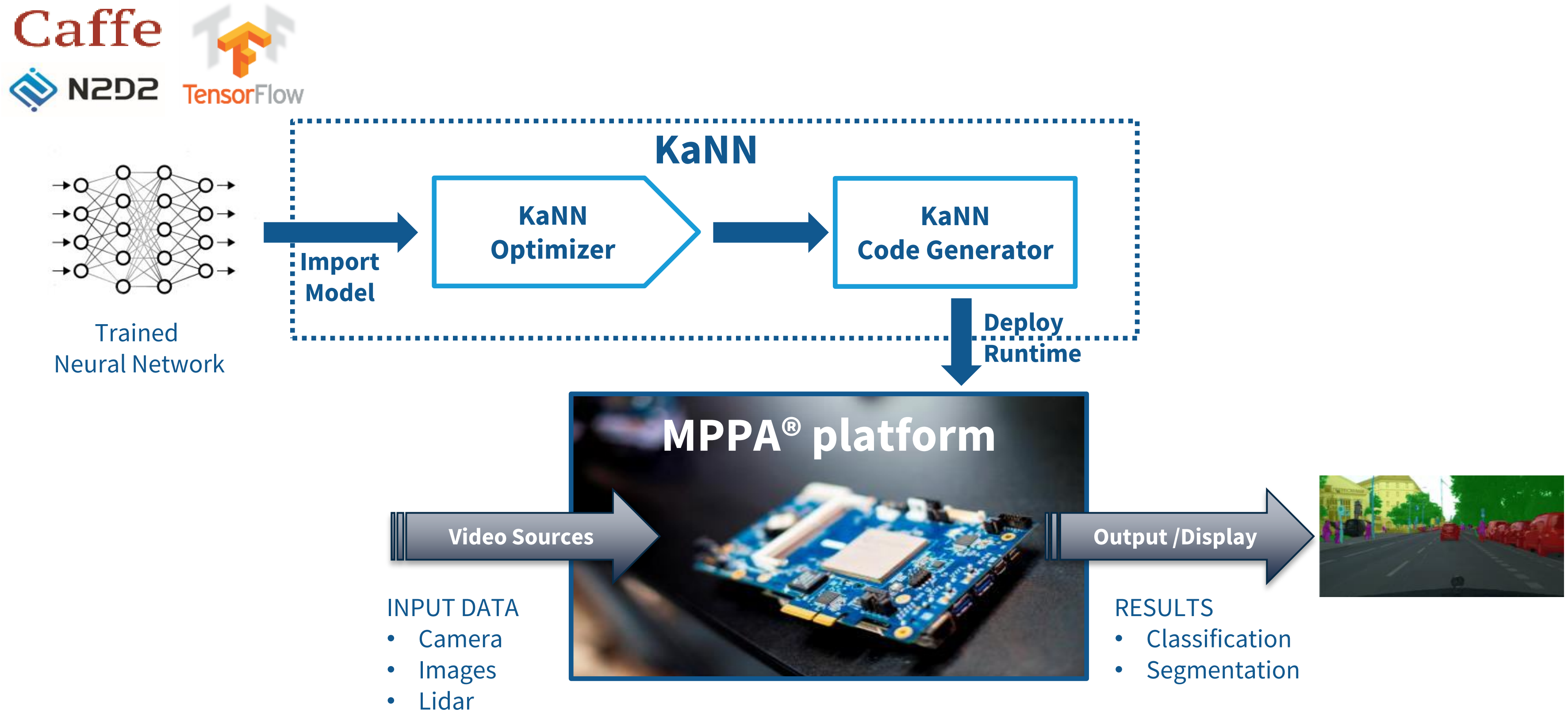
YOLO v1-3 « You Only Look Once » (Redmon 2016-2018)

Single-step method (contrairement aux « R-CNN »)

- *Input image is processed only once by the network*
- *Approximate localization of small objects in clusters*



KaNN: Kalray CNN Inference Code Generator



Caffe

N2D2

TensorFlow

Trained
Neural Network

Import
Model

KaNN

KaNN
Optimizer

KaNN
Code Generator

Deploy
Runtime

MPPA® platform

Video Sources

Output /Display

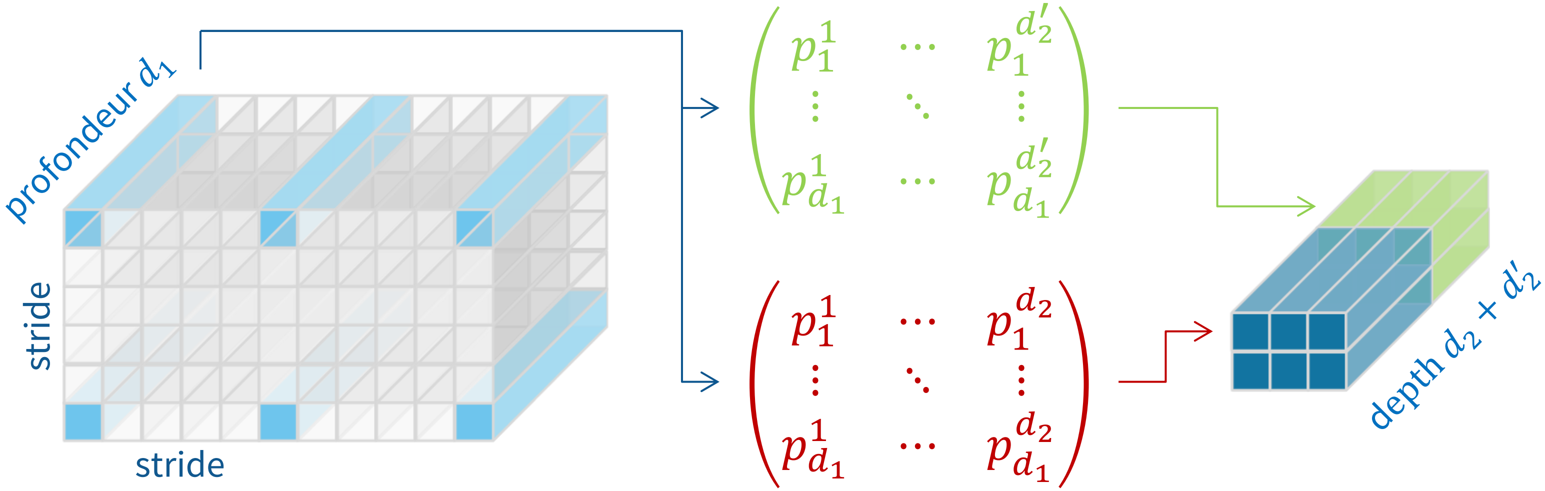
INPUT DATA
- Camera
- Images
- Lidar

RESULTS
- Classification
- Segmentation

CNN Inference on a MPPA® Processor (1)

NxN convolutions decomposed as accumulations of N^2 1x1 convolutions

- 1x1 convolutions can be computed in parallel and accumulated in any order
- Pixels layout is sequential along depth (channels) for dense memory accesses



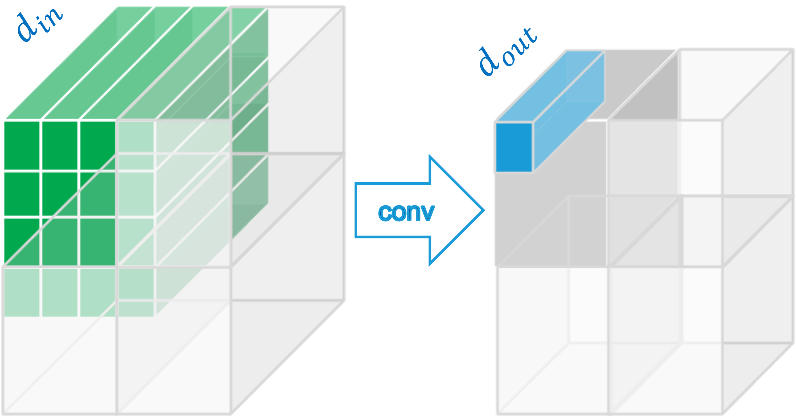
$$\begin{pmatrix} p_1^1 & \cdots & p_1^{d_2'} \\ \vdots & \ddots & \vdots \\ p_{d_1}^1 & \cdots & p_{d_1}^{d_2'} \end{pmatrix}$$

$$\begin{pmatrix} p_1^1 & \cdots & p_1^{d_2} \\ \vdots & \ddots & \vdots \\ p_{d_1}^1 & \cdots & p_{d_1}^{d_2} \end{pmatrix}$$

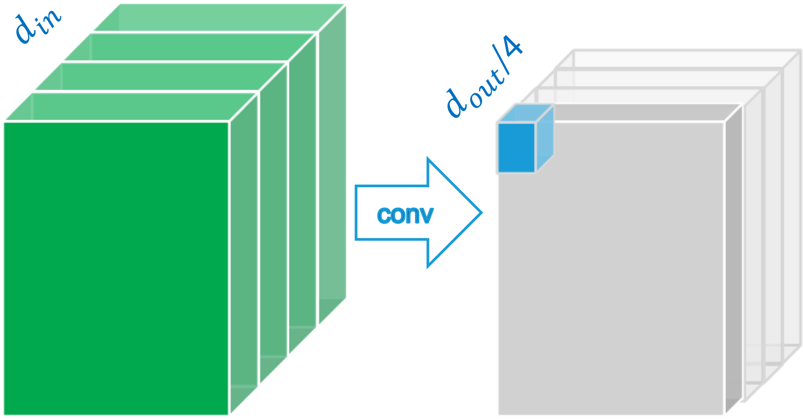
CNN Inference on a MPPA® Processor (2)

Partition images across clusters, splitting along spatial and/or depth dimensions

- Spatial dimension splitting requires that the full set of parameters be loaded from external memory
- Channel dimension splittig requires access to the whole input image and a subset of the parameters
- NoC multicasting of parameters fosters spatial dimension splitting except for small dimensions (e.g. FC)



$[3][3][d_{in}][d_{out}]$



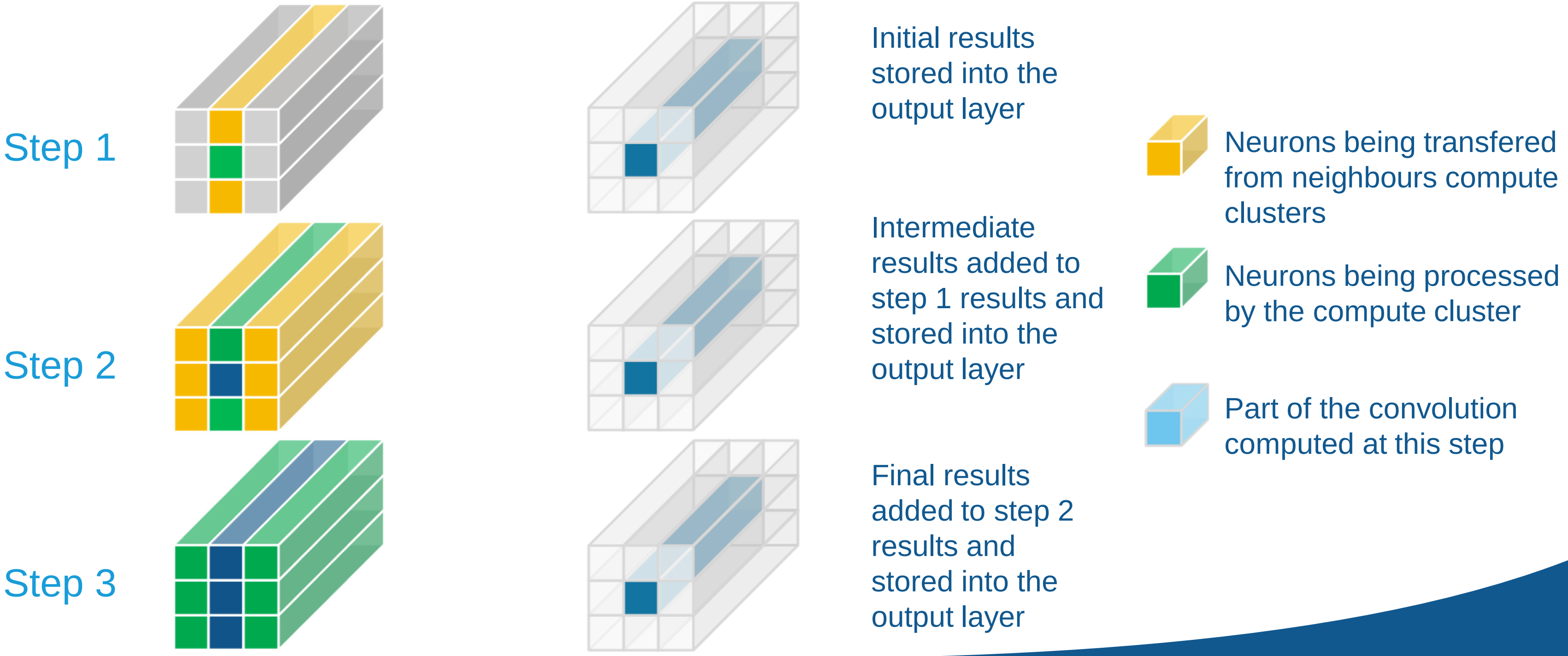
$[3][3][d_{in}][d_{out}/4]$

CNN Inference on a MPPA® Processor (3)

Process layers sequentially, distributing computations across all available clusters

Each cluster local memory stores a tile + shadow region of the previous layer

Compute the current layer in 3 steps to overlap with shadow region transfers



Step 1

Initial results stored into the output layer

Step 2

Intermediate results added to step 1 results and stored into the output layer

Step 3

Final results added to step 2 results and stored into the output layer

🟨 Neurons being transfered from neighbours compute clusters

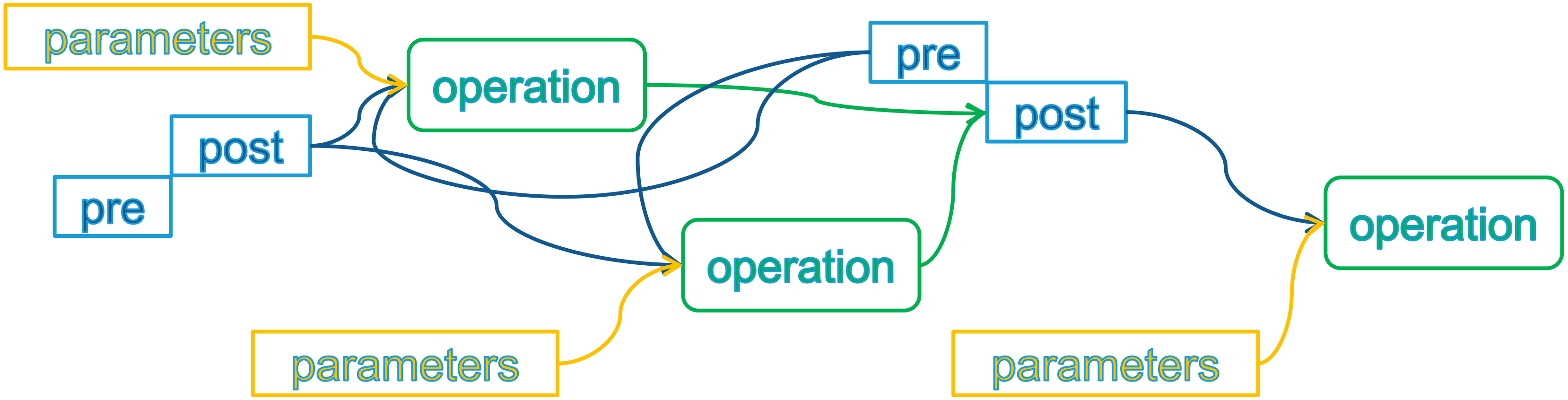
🟩 Neurons being processed by the compute cluster

🟦 Part of the convolution computed at this step

CNN Inference on a MPPA® Processor (4)

Build a buffer allocation and task execution schedule in cluster memory to overlap parameter transfers from external memory with computations on local memory

Allocation and scheduling are performed on the CNN network, considering an image correspond to pre and post tasks, and computations correspond to a malleable task

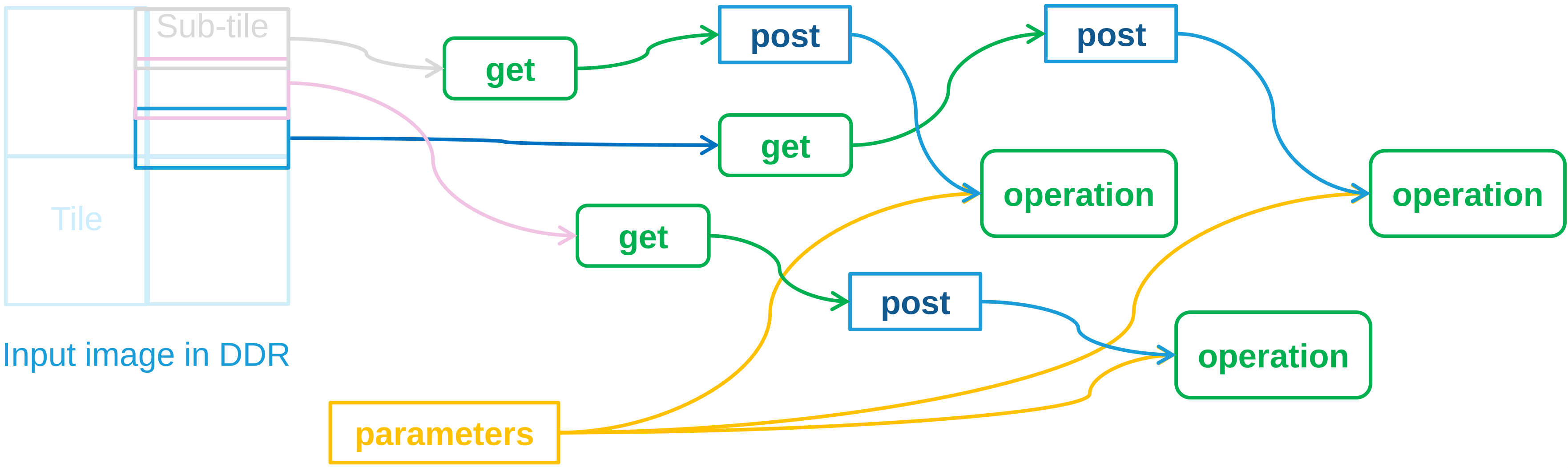


On MPPA® processors, parameter loading from DDR leverages NoC multicasting

CNN Inference on a MPPA® Processor (5)

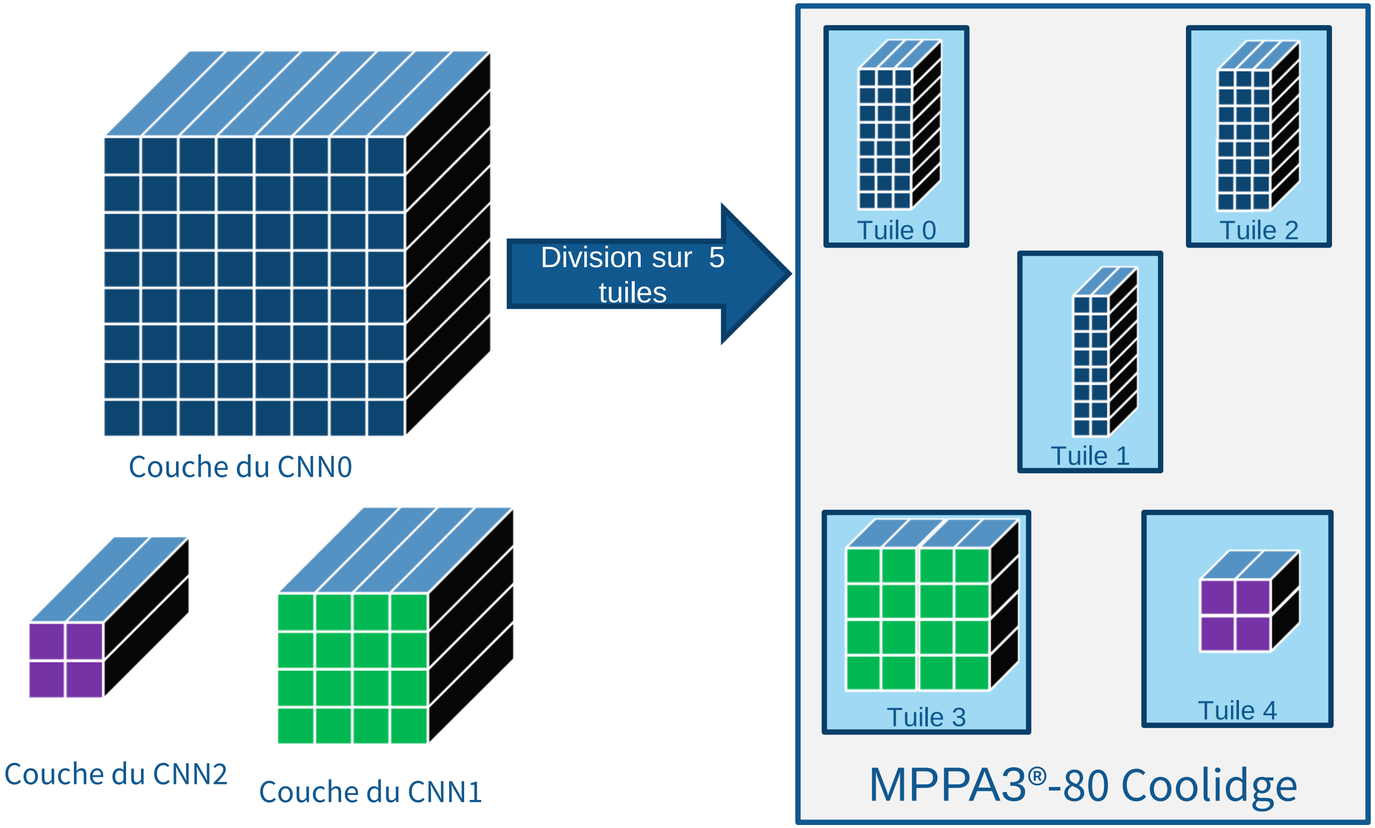
For layers where images do not fit on-chip, stream sub-tiles from DDR memory

- All clusters remote write their tile of output image to DDR memory, then enter a synchronization barrier
- After clusters leave the barrier, they pipeline the remote read from DDR / operate / put to DDR of sub-tiles
- Larger sub-tiles factor more control overhead but reduce the amount of pipelining

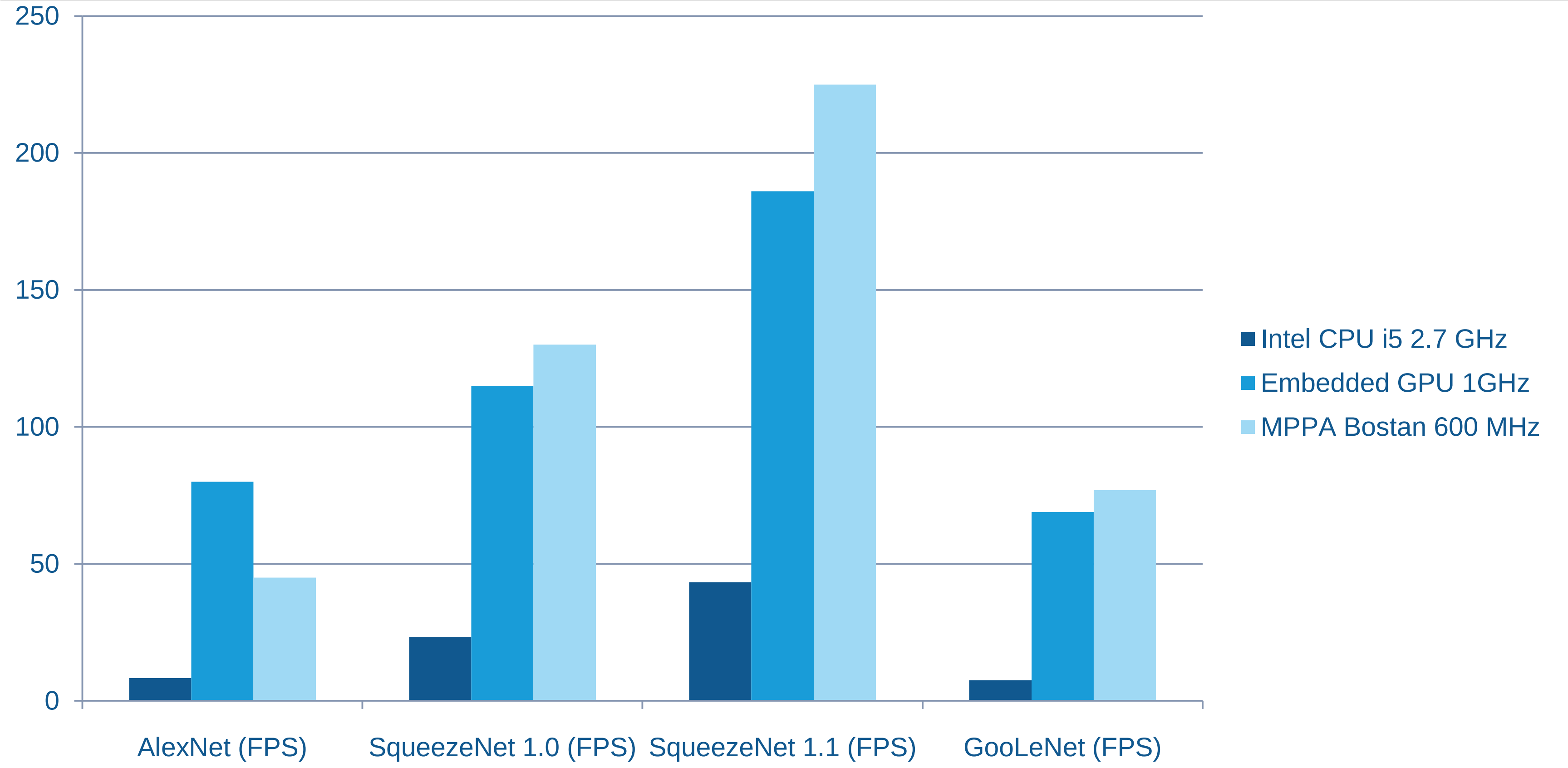


Multiple CNN Inferences on the MPPA®-80 Coolidge

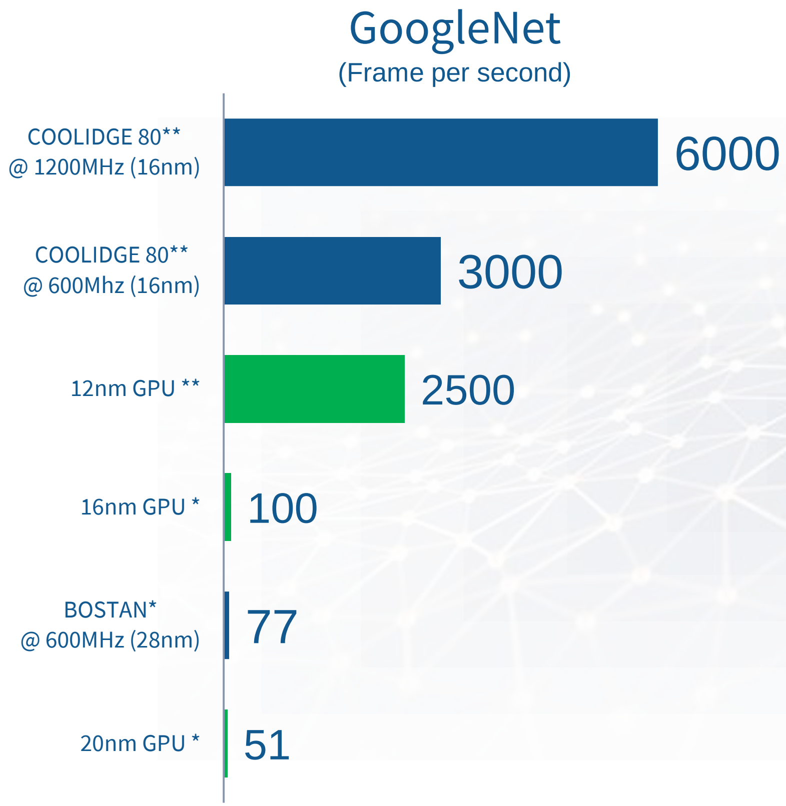
Exécution de réseaux multiple par partitionnement spatial du MPPA



MPPA® Bostan vs CPU & GPU on CNN Inference



MPPA®: A PROCESSOR FOR DEEP LEARNING



(*) Measurement
(**) Estimation

MPPA processors are especially well-suited for efficient deep learning and computer vision

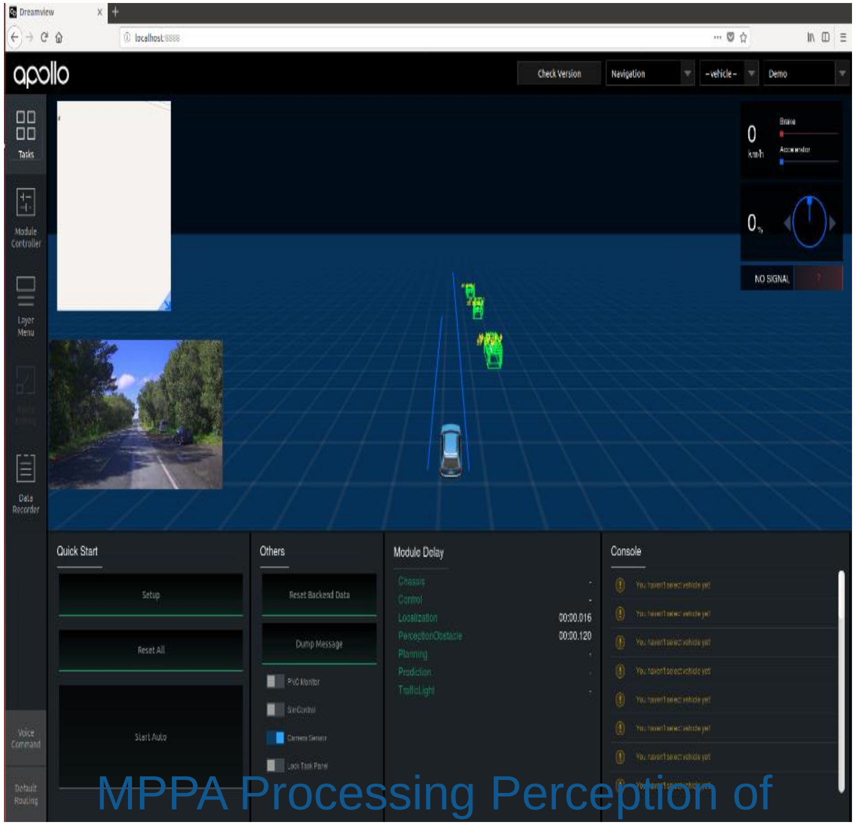
- Specific Co-processor for Vision and Learning
  - 16-bits floats for more than 3 TFLOPS
  - 8-bits fixed point for up to 6TFLOPS
- High on chip memory bandwidth 300GB/s to store data closer to the compute units
- Fast and direct communication between clusters and chip for faster communication between layers

MPPA® DEEP LEARNING PERFORMANCES

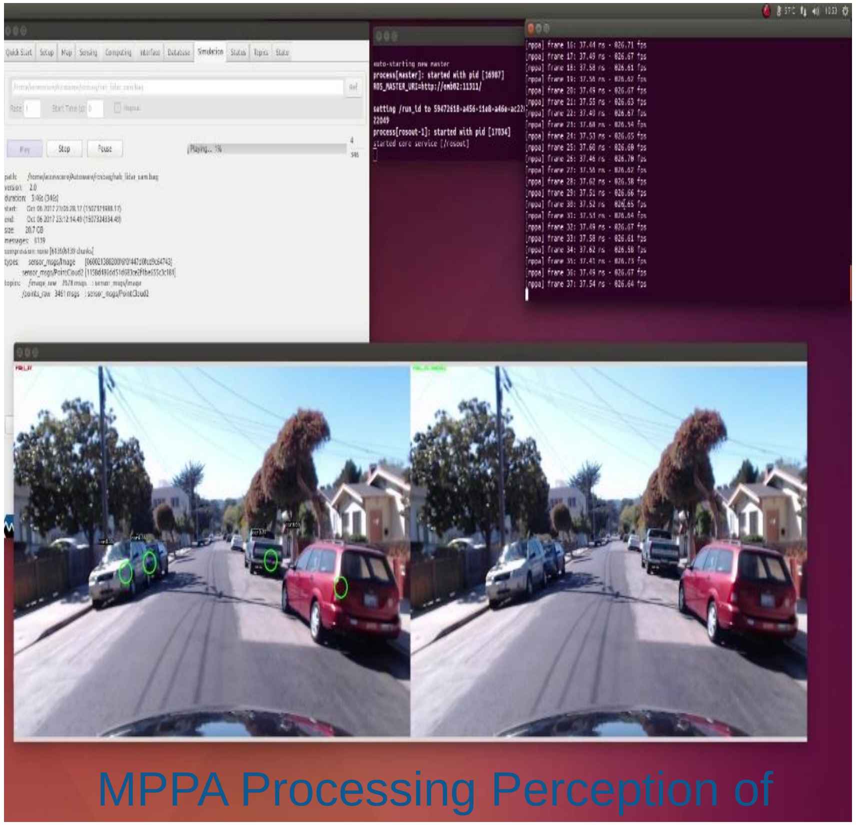
	Bostan @500MHz	Coolidge-80 v1 @1.2 GHz	Coolidge-80 v2 @1.2 GHz
GoogleNet	65 fps (FP32)*	1500 fps (INT16)** 3000 fps (INT8)**	3000 fps (INT16)** 6000 fps (INT8)**
SqueezeNet 1.1	218 fps (FP32)*	4950 fps (INT16)** 9900 fps (INT8)**	9900 fps (INT16)** 19800 fps (INT8)**
SqueezeNet 1.0	106 fps (FP32)*	2610 fps (INT16)** 5220 fps (INT8)**	5220 fps (INT16)** 10440 fps (INT8)**
VGG-16	7 fps (FP32)*	180 fps (INT16)** 360 fps (INT8)**	360 fps (INT16)** 720 fps (INT8)**
ResNet-50	35 fps (FP32)*	870 fps (INT16)** 1740 fps (INT8)**	1740 fps (INT16)** 3480 fps (INT8)**

(*) Measurements of computing on MPPA®
(**) Estimation based on simulation and results from Bostan

KaNN Integration into 3rd Party Autonomous Software Platforms



MPPA Processing Perception of
BAIDU Apollo



```
auto-starting new master
process[master]: started with pid [16987]
ROS_MASTER_URI=http://emb02:11311/

setting /run_id to 59472618-a456-11e8-a46e-ac22
22049
process[rosout-1]: started with pid [17034]
started core service [/rosout]
```

```
[mppa] frame 16: 37.44 ms · 26.71 fps
[mppa] frame 17: 37.49 ms · 26.67 fps
[mppa] frame 18: 37.58 ms · 26.61 fps
[mppa] frame 19: 37.56 ms · 26.62 fps
[mppa] frame 20: 37.49 ms · 26.67 fps
[mppa] frame 21: 37.55 ms · 26.63 fps
[mppa] frame 22: 37.49 ms · 26.67 fps
[mppa] frame 23: 37.68 ms · 26.54 fps
[mppa] frame 24: 37.53 ms · 26.65 fps
[mppa] frame 25: 37.60 ms · 26.60 fps
[mppa] frame 26: 37.46 ms · 26.70 fps
[mppa] frame 27: 37.56 ms · 26.62 fps
[mppa] frame 28: 37.62 ms · 26.58 fps
[mppa] frame 29: 37.51 ms · 26.66 fps
[mppa] frame 30: 37.52 ms · 26.65 fps
[mppa] frame 31: 37.54 ms · 26.64 fps
[mppa] frame 32: 37.49 ms · 26.67 fps
[mppa] frame 33: 37.58 ms · 26.61 fps
[mppa] frame 34: 37.62 ms · 26.58 fps
[mppa] frame 35: 37.41 ms · 26.73 fps
[mppa] frame 36: 37.49 ms · 26.67 fps
[mppa] frame 37: 37.54 ms · 26.64 fps
```

MPPA Processing Perception of
Autoware

Outline

Presentation

Manycore Processors

Manycore Programming

Symmetric Parallel Models

Untimed Dataflow Models

Kalray MPPA® Hardware

Kalray MPPA® Software

Model-Based Programming

Deep Learning Inference

Conclusions

Consolidating the MPPA® Eco-System

Model Based Programming

ANSYS

ESTEREL Technologies

MATLAB SIMULINK

Standard Programming

OpenVX

C/C++

OpenCL

Academic Partners

Inria

ONERA

cea

Certification Partners

onespin

TÜV SÜD

iROC TECH

Libraries

Caffe

TensorFlow

N2D2

FFTW

OpenCV

BLAS

pointcloudlibrary

Operating Systems

Linux

ROS

eSOL

ERIKA ENTERPRISE

KRONO-SAFE

PROVE & RUN

Automotive Platforms

apollo

AUTOWARE

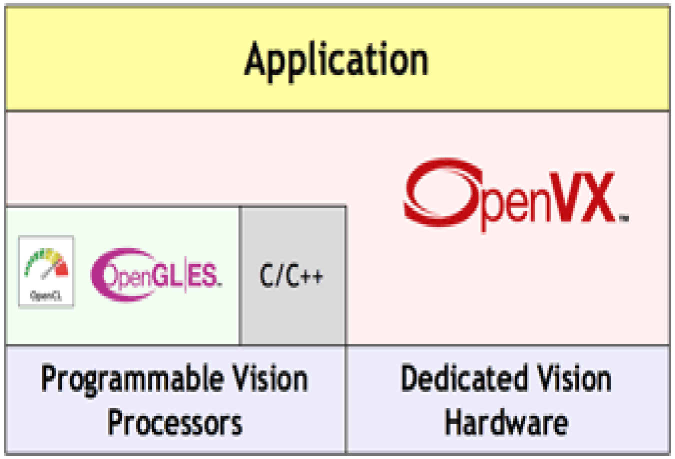
Khronos OpenVX for Computational Imaging

OpenVX express a graph of image operations (‘Nodes’)

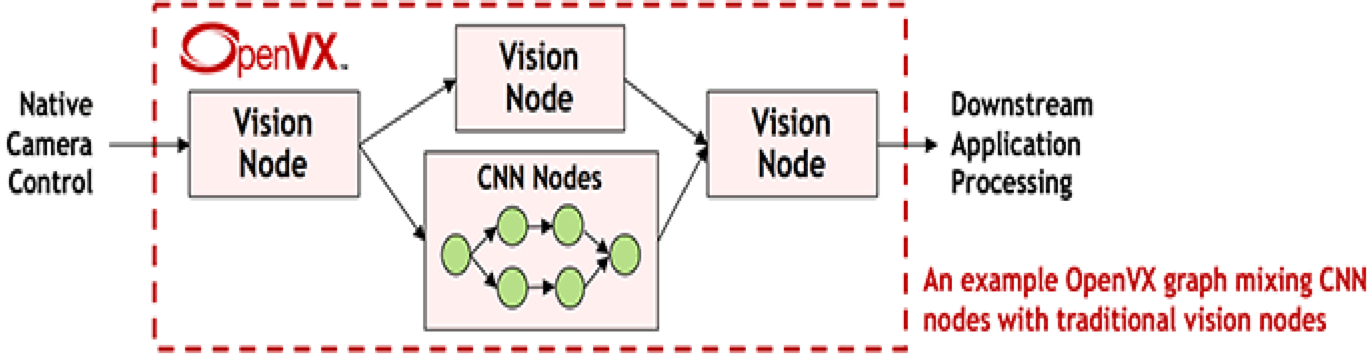
- Nodes can be run on any hardware or processor and coded in any language

Graph-based computing enables implementations to optimize for power and performance

- Nodes may be fused by the implementation to eliminate memory transfers
- Processing can be tiled to keep data entirely in local memory/cache

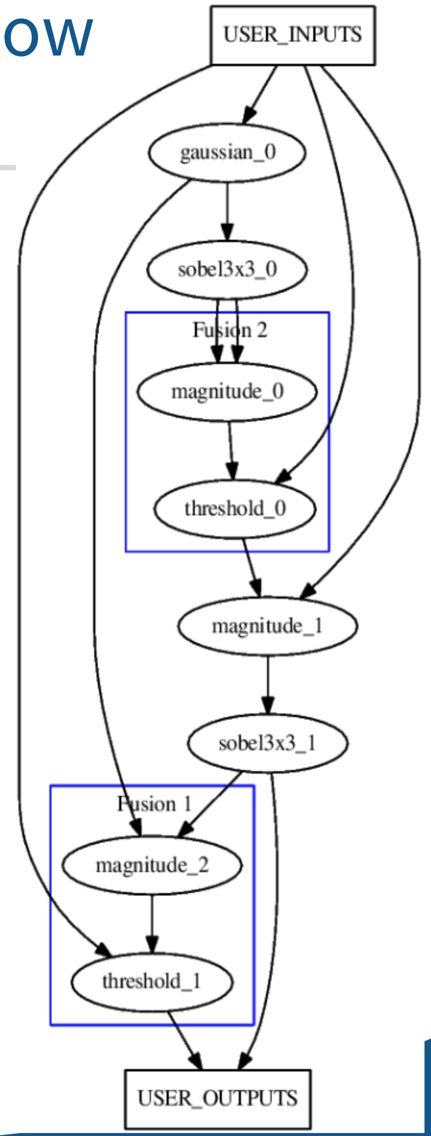


Minimizes host interaction during frame-rate graph execution



An example OpenVX graph mixing CNN nodes with traditional vision nodes

Kalray OpenVX Compilation Workflow



Build and check the Single-Rate Directed Acyclic Graph

- No multi-writer on outputs
- No unconnected image buffers
- At least one user input and output

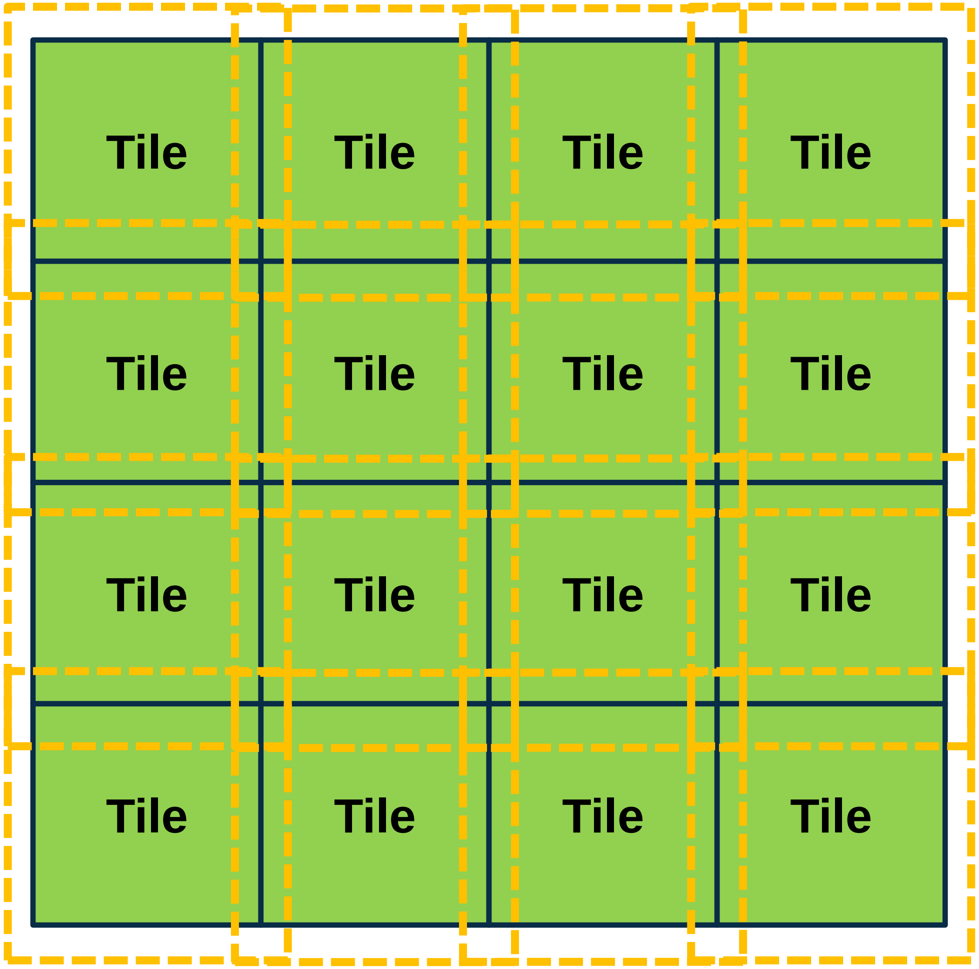
Detect kernel fusion opportunities on virtual images

- Pairwise grouping of adjacent nodes
- Local memory capacity constraints
- Kernel dependency pattern
- Edge type (real or virtual)

Code generation for SPMD execution

- Topological sort scheduling of nodes
- Build allocation plan for local memory buffers
- Select commands for tiling/skewing runtime engines

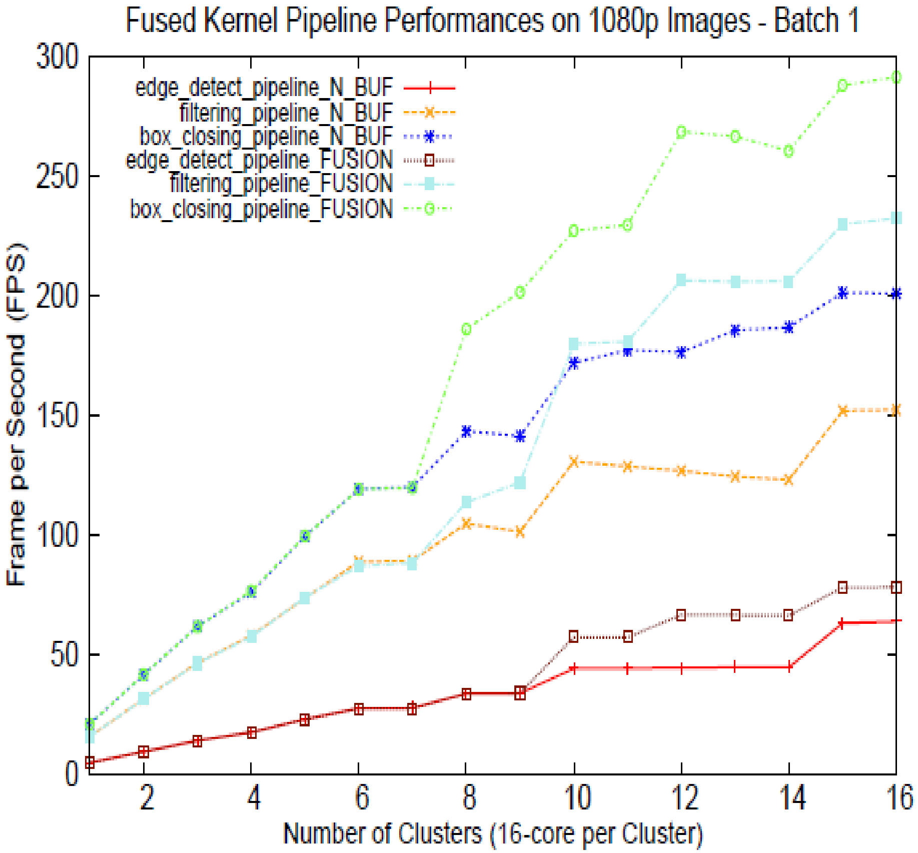
Kalray OpenVX N-Buffering Tiling Engine

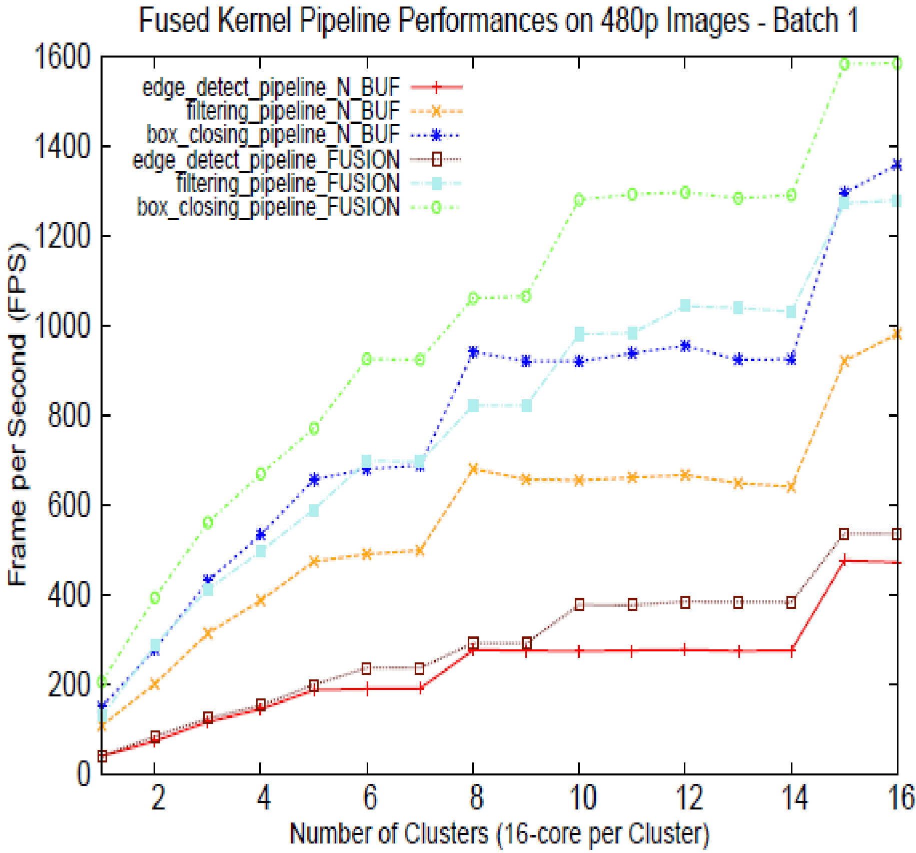


```
/* prologue */
for i in 0 .. N-1
    get(i)
/* kernel */
for(i in N-1 .. NB_TILE)
    wait(i – (N-1))
    kernel(i – (N-1))
    put(i – (N-1)) // write results
    get(i) // prefetch
/* epilogue */
for(i in NB_TILE-(N-1) .. NB_TILE)
    kernel(i)
    put(i)
fence()
```

MPPA2® Bostan Performances on OpenVX

Automated kernel fusion in MPPA2® OpenVX environment


Fused Kernel Pipeline Performances on 1080p Images - Batch 1


Fused Kernel Pipeline Performances on 480p Images - Batch 1

Conclusions and Perspectives

The MPPA® manycore architecture excels on standard CNN inference

- Not only on performance, but also on energy efficiency and time-predictability
- The key is to exploit the high-bandwidth local memory shared by cores in a cluster
- This is achieved by the KaNN code generation tool working from standard frameworks

Techniques applied by the KaNN code generator are generalized

- KaNN extensions to 8-bit/16-bit fixed-point inference as supported by standard frameworks (TensorFlow gemmlowp, Caffe Ristretto)
- OpenVX framework for MPPA® processors to be released in 2019

Standard OpenCL environment must be extended

- OpenCL Task Parallel mode extensions to support C/C++, pthreads & OpenMP, and asynchronous one-sided operations between Compute Units (MPPA® compute clusters)

Model-based execution environments

- Model-based environments (SCADE, Simulink) unlocks use of manycore processors
- Further developments that combine SCADE Suite (Esterel) and Asterios (Krono-Safe)

MPPA® Technology

SAFETY

- Hardware partitioning
- Software partitioning
- Hypervisor support
- ISO26262 ASIL B/C

SECURITY

- Hardware root of trust
- Secure boot
- Authenticated debug
- Trusted execution environment
- Encrypted application code

DETERMINISM

- Fully timing compositional cores
- Banked on-chip memory
- Interference-free local interconnect
- Network-on-Chip (NoC) service guarantees

PERFORMANCE

- High-end floating-point and bit-level processing
- DSP-style energy efficiency
- Scalability by replicating clusters

STANDARDS

- Standard programming environments (C/C++, OpenMP, POSIX, OpenCL, OpenVX)
- Standard development tools (Eclipse, GCC, GDB, LLVM, Linux)

SCALABLE

- Adaptability to E/E architecture
- Low range to high range car lines
- Allow distribution of functions

KALRAY S.A. - GRENOBLE - FRANCE
180 avenue de l’Europe,
38 330 Montbonnot - France
Tel: +33 (0)4 76 18 09 18
email: info@kalray.eu

KALRAY INC. - LOS ALTOS - USA
4962 El Camino Real
Los Altos, CA - USA
Tel: +1 (650) 469 3729
email: info@kalrayinc.com

MPPA, ACCESSCORE and the Kalray logo are trademarks or registered trademarks of Kalray in various countries. All trademarks, service marks, and trade names are the marks of the respective owner(s), and any unauthorized use thereof is strictly prohibited. All terms and prices are indicative and subject to any modification without notice.

KALRAY