Dataflow Language Compilation for a Single Chip Massively Parallel Processor

MuCoCoS 2013
September 7, 2013, Edinburgh, Scotland, UK

Benoît Dupont de Dinechin
Outline

- Kalray MPPA® Products
  - Kalray SigmaC Language Features
  - Kalray SigmaC Language Toolchain
  - Related Dataflow Languages and Toolchains
  - Kalray MPPA® Application Examples
- Conclusions
Kalray MPPA® Products

High performance, low power single-chip massively parallel processors

C/C++ based Software Development Kit (SDK) for massively parallel programming

Development platform Reference Design Board
Kalray MPPA®-256 Processor with CMOS 28nm TSMC

256 VLIW processing engine cores + 32 VLIW resource management cores

- High processing performance
  700 GOPS – 230 GFLOPS SP
- Low power consumption
- High execution predictability
- High-level programming models
- PCI Gen3, Ethernet 10G, NoCX

Available since November 2012
MPPA MANYCORE Processor Roadmap

Architecture scalability for high performances and low power

**MPPA®-256 V1**
- 1st core generation
- 50 GFLOPS/W

**MPPA®-64**
- Low Power
- 12W

**MPPA®-256 V2**
- 2nd core generation
- 80 GFLOPS/W

**MPPA®-1024**
- Very Low Power
- 1.8W / 0.6W
- Idle 75mW

**2012**
- 28nm

**2014**
- 28nm → 20nm

**2015**
- 16nm
MPPA®-256 Processor Hierarchical Architecture

- VLIW Core
- Compute Cluster
- Manycore Processor

- Instruction Level Parallelism
- Thread Level Parallelism
- Process Level Parallelism
MPPA®-256 VLIW Core Architecture

- Data processing code
  - Byte memory alignment
  - Standard & effective FPU
  - Configurable bitwise logic
  - Hardware looping

- System & control code
  - MMU → single memory port → no function unit clustering

- Execution predictability
  - Fully timing compositional core
  - LRU caches, low miss penalty

- Energy and area efficiency
  - 7-stage instruction pipeline, 400MHz
  - Idle modes and wake-up on interrupt

- 5-issue VLIW architecture
- Predictability & energy efficiency
- 32-bit/64-bit IEEE 754 FPU
- MMU for rich OS support
MPPA®-256 Compute Cluster

- Multi-banked parallel memory
  - 38.4GB/s of bandwidth @400MHz
- Reliability
  - ECC in the shared memory
  - Parity check in the caches
  - Faulty cores can be switched off
- Predictability
  - Multi-banked shared memory with interleaved or blocked address map
- Low power
  - Memory banks with low power mode
  - Voltage scaling

- 16 PE cores + 1 RM core
- NoC Tx and Rx interfaces
- Debug Support Unit (DSU)
- 2 MB of shared memory
MPPA®-256 Clustered Memory Architecture

- 20 memory address spaces
  - 16 compute clusters
  - 4 I/O subsystems with direct access to external DDR memory

- Dual Network-on-Chip (NoC)
  - Data NoC & Control NoC
  - Full duplex links, 4B/cycle
  - 2D torus topology + extension links
  - Unicast and multicast transfers

- Data NoC QoS
  - Flow control and routing at source
  - Guaranteed services by application of network calculus
  - Oblivious synchronization
Source traffic regulation using \((\sigma, \rho)\)

- A packet flow obeys \((\sigma, \rho)\) if for any time interval \(\tau\) the number of packets is not greater than \(\sigma + \rho \tau\)
- The initial \((\sigma, \rho)\) is set at the Tx Data NoC interface
MPPA®-256 Processor I/O Interfaces

- DDR3 Memory interfaces
- PCIe Gen3 interface
- 1G/10G/40G Ethernet interfaces
- SPI/I2C/UART interfaces
- Universal Static Memory Controller (NAND/NOR/SRAM)
- GPIOs with Direct NoC Access (DNA) mode
- NoC extension through Interlaken interface (NoC Express)
MPPA® Architecture Compared to other Manycores

- NVIDIA, ATI, ARM generalize the GPU architecture into GP-GPU
  - Streaming multiprocessors that share a cache and DDR memory
  - Each stream multiprocessor operates multi-threaded cores in SIMT
  - CUDA or OpenCL data parallel kernel programming models

- Cavium, Tilera TILE Gx, Intel MIC support shared coherent memory
  - Thread-based parallel programming (POSIX threads, OpenMP)
  - Non uniform memory access (NUMA) times, challenging cache design

- Kalray MPPA® extends the supercomputer clustered architecture
  - Clustered memory architecture scales to > 1M cores (BlueGene/Q)
  - Low energy per operation, high execution predictability
  - Stand-alone operation with I/O, low-latency processing
Kalray Software Development Kit
MPPA ACCESSCORE – MPPA ACCESSLIB

Today

Q4 2013

This talk

Standard C/C++ Programming Environment

Dataflow Programming FPGA Style

Simulators, Profilers, Debuggers & System Trace

POSIX-Level Programming DSP Style

Operating Systems & Device Drivers

Streaming Programming GPU Style
Dataflow Models of Computation

- Kahn Process Networks (KPN) [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 firing sequence
Dataflow Models of Computation

- Dataflow Process Networks (DPN) [Lee & Parks 1995]
  - KPN with functional actor firing (no persistent agent state)
  - KPN with sequential firing rules (can be tested in a pre-defined order using only blocking reads)

- Synchronous Dataflow [Benveniste et al. 1994]
  - Clocks are associated with tokens carried by the channels

- Static Dataflow (SDF) [Lee & Messerschmitt 1987]
  - Agents producing and consuming a constant number of tokens
  - Single-rate SDF is known as Homogenous SDF (HSDF)

- 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
Dataflow Programming Environment

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

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

- Kalray MPPA® Products
- **Kalray SigmaC Language Features**
- Kalray SigmaC Language Toolchain
- Related Dataflow Languages and Toolchains
- Kalray MPPA® Application Examples
- Conclusions
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();
    }
}
Example of Cyclostatic Specs

```c
spec{(5){input}; {input; output}};

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

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

void start ()
{
    int i,
    for (i=0; i<5; i++) {
        fn1();
    }
    fn2();
}
```

5 input transitions before processing another input and firing output
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

Send every other input to a different output (split)

Consume one input at a time
At program startup, some channels may be non-empty
  - Required for the liveness of some dataflow graphs

```c
void preload(input_channel, int token_nbr, int data_size, void *input_data);
```
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)
{
}

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

<table>
<thead>
<tr>
<th>Accessible in 1st transition</th>
<th>Consumed in 1st transition</th>
<th>Accessible in 2nd transition</th>
<th>Consumed in 2nd transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>input stream</td>
<td></td>
</tr>
</tbody>
</table>
### Instanciating and Connecting Agents

- **The « map » section of « subgraphs »**

```java
subgraph Subgraph1 ()
{
    interface
    {
        in<unsigned char> input;
        out<unsigned char> output;
    }
    spec{input; output};
}

map {
    int N = 1024;
    agent a1 = new Agent1();
    agent a2 = new Agent2(N);
    connect(input, a1.input);
    connect(a1.output, a2.input);
    connect(a2.output, output);
}
```

- Agents are instanciated via the « new » keyword
- « N » is an instance parameter for the agent
- Agent interfaces are connected using « connect »
Split and ComplexSplit System Agents

Split: uniform scatter to N consumers

agent Split<elt_type>(int N, int k) {
    interface {
        in<elt_type> input;
        out<elt_type> output[N];
        spec {{input[k]; output[0][k]}; ...; {input[k]; output[N-1][k]}};
    }
}

ComplexSplit: cyclic scatter to N consumers

agent ComplexSplit<elt_type>(int N, unsigned int p, const unsigned int coefs[p]) {
    interface {
        in<elt_type> input;
        out<elt_type> output[N];
        spec {{input[coefs[0]]; output[0][coefs[0]]}; ...; {input[coefs[p-1]]; output[p-1][coefs[p-1]]}; {input[coefs[0]]; output[p][coefs[0]]}; ... };
    }
}
Join and ComplexJoin System Agents

Join: uniform gather from N producers

agent Join<elt_type>(int N, int k) {
  interface {
    in<elt_type> input[N];
    out<elt_type> output;
    spec {{input[0][k]; output[k]}; ...;
          {input[N-1][k]; output[k]}};
  }
}

ComplexJoin: cyclic gather from N producers

agent ComplexJoin<elt_type>(int N, unsigned int p, const unsigned int coefs[p]) {
  interface {
    in<elt_type> input[N];
    out<elt_type> output;
    spec {{input[0][coefs[0]]; output[coefs[0]]}; ...;
          {input[p-1][coefs[p-1]]; output[coefs[p-1]]};
          {input[p][coefs[0]]; output[coefs[0]]}; ... }
  }
}
**Dup and Sink System Agents**

**Dup: broadcast to N consumers**

```c
agent Dup<elt_type>(unsigned N, unsigned k) {
    interface {
        in<elt_type> input;
        out<elt_type> output[N];
        spec {{input[k]; output[]}[k]};
    }
}
```

**Sink: throw away tokens**

```c
agent Sink<elt_type>(unsigned int k) {
    interface {
        in<elt_type> input;
        spec {input[k]};
    }
}
```
System Agent Inlining

- Inlining saves buffer memory space and runtime copies
  - Requires that read and writes occur in middle of buffer
  - Data communication primitives are not FIFO operations

- Inlining constrained by the ‘pointer equivalence’ principle
  - User code inside agents use regular pointers to access tokens
Outline

- Kalray MPPA® Products
- Kalray SigmaC Language Features
- Kalray SigmaC Language Toolchain
- Related Dataflow Languages and Toolchains
- Kalray MPPA® Application Examples
- Conclusions
Static Dataflow Graph Boundedness

- **Balance equations**
  - $2N(A) - N(B) = 0$
  - $N(B) - N(C) = 0$
  - $2N(A) - N(C) = 0$
  - $2N(A) - N(C) = 0$

- **Graph incidence matrix**

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

- **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 execution**
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

\[
\begin{array}{c}
A \rightarrow 3 \rightarrow B \rightarrow 3 \rightarrow C
\end{array}
\]

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

  - True data dependence arcs and buffer size feedback arcs

\[
\begin{array}{c}
A \rightarrow \ldots \rightarrow B \rightarrow \ldots \rightarrow C
\end{array}
\]
Dataflow Compilation and Execution Overview

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
Phase 1

- Parse Sigma-C source files
  - Flex / Bison lexer-parser
    - Accept C99 + GNU C extensions
  - Resolve templating of agents

- C code generation
  - Generate code for instanciation of dataflow graph
  - Generate code for agents local data and functions
    - Leverage nested functions of GNU C
  - Insert buffer access macros in agent code
    - Allow late changes to buffer implementation
Phase 2

- Dataflow graph construction
  - Compile and execute map sections on toolchain host

- Dataflow graph coherency checks
  - Ensure there are no dangling ports
  - Check token structure compatibility between execution targets

- Produce intermediate representation
  - Flatten the dataflow graph
  - Compute channel initial tokens values
  - Resolve agent instance parameters to constants
Phase 3

- Balance equations
  - Find agent periods $N_i$ (hyperperiod)
  - Replicate graph to consume initial tokens ($k_1$-hyperperiod)

- System agent inlining analysis
  - Check that pointer equivalence is maintained
  - Compute minimum sizes of inlined buffers

- First symbolic execution
  - Compute minimum buffer sizes for liveness of dataflow graph
  - Build the generic data precedence graph

- (Advanced cyclostatic dataflow sizing and sequencing)
Phase 3

- Second symbolic execution
  - Compute k2-hyperperiod that activates the buffer feedback arcs
- Inlining of system agents
  - Resize the inlined buffers
- Pad buffers and insert shadow copy code
  - Maintain pointer equivalence with preloads and thresholds
- Mapping of tasks to platform resources
  - Use simulated annealing with placement constraints
  - Check effects on buffer sizes and inlining decisions
- Routing over NoC, PCIe and Ethernet
  - Compute routes and source flow restrictions
Phase 3

- **Runtime generation**
  - Compute buffer pointer increments
  - Generate dependency descriptors for runtime engine
  - Generate NoC configuration bit-stream
  - Compute FIFO sizes for inter-cluster dependency descriptors

- **Dataflow graph rewriting**
  - Map non-inlined system agents to DMA tasks
  - Coalesce inter-cluster transfers
  - Combine system agents

- Any dataflow graph rewriting restarts P3
Sigma-C Toolchain Targets

- Native simulator
  - Self scheduled
    - Synchronised by channel read/write
  - Sequential
    - Only one agent active at a time for debug purposes
  - Sequenced
    - Synchronisation via a pre-computed partial order of P3

- Mixed simulator
  - Native simulation engine running on host
  - Agents compiled to VLIW core instruction set and run on ISS

- Multicores and manycore
  - X86_32, x86_64, MPPA platforms
Outline

- Kalray MPPA® Products
- Kalray SigmaC Language Features
- Kalray SigmaC Language Toolchain
- Related Dataflow Languages and Toolchains
- Kalray MPPA® Application Examples
- Conclusions
Ptolemy II (Berkeley)

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

example Ptolemy II model: hybrid control system

http://ptolemy.eecs.berkeley.edu
Filters are autonomous unit of computation
- No global resources
- FIFO channels
  - `pop()` / `peek(index)` / `push(value)`
- Peek / pop / push rates must be constant

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

Teleport messaging
- Program morphing
- RAW machine code generation

Pipe-line may be any StreamIt language construct

Parallel computation

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

- Kalray MPPA® Products
- Kalray SigmaC Language Features
- Kalray SigmaC Language Toolchain
- Related Dataflow Languages and Toolchains
- Kalray MPPA® Application Examples
- Conclusions
MPPA®-256 PCIe Application Board AB01

- Connect to the 4 I/O subsystems
  - 2 PCIe GEN3 x8 interfaces through a x16 PCIe switch
  - 2 DDR3 interfaces
  - 4 Ethernet interfaces (2x10G + 2x1G)
  - 4 Interlaken interfaces
  - NOR flash, GPIOs, leds, buttons, extensions & debug connectors
Video Broadcasting Demonstrator

- High definition H264 encoder on one MPPA®-256 processor
- System integration, lower power and cost
- Intel CPU + MPPA® implementation
- Flexibility & scalability

H264 encoder running on MPPA®-256 at less than 6W
Dataflow H264 Encoder on the MPPA®-256 processor

- Better quality (SSIM and PSNR criteria) than C reference
  - Additional motion vectors and intra predictors tested (in parallel) without throughput impact.
- Intra I-frame: 110 fps.
- Inter P-frame: 40 fps.
- Inter B-frame: 55 fps.
Signal Processing Examples

- Radar applications: STAP, ...
- Beam forming: Sonar, Echography
- Software Defined Radio (SDR)
- Dedicated libraries (FFT, FTFR, ...)

Well suited to massively parallel architectures
Alternative of embedded DSP + FPGA platforms
Panoramic surveillance with a linear antenna
- 3 very low frequency antennas sampled at 3840 Hz
  - [640, 1280 Hz] : 144 hydrophones
  - [320, 640 Hz] : 144 hydrophones
  - [160, 320 Hz] : 144 hydrophones
- Compute 180 beams per antenna

Sonar Beam Forming

- Preprocess TBF1 → beamform → FFT
- Preprocess TBF2 → beamform → FFT
- Preprocess TBF3 → beamform → FFT
- Join
Beamform Dataflow Graph

- Subgraph for one antenna

Sensor sample → Split → ½ Circular Buffer → Beam #1 → Sum #1 → Beam sample

Sensor sample → Split → ½ Circular Buffer → Beam #13 → Sum #13 → Beam sample

Fit in cluster memory

Meet profiling target
Application Performance Analysis

Agent execution waveform

Agent effective parallelism
Application Mapping Analysis
Outline

- Kalry MPPA® Products
- Kalray SigmaC Language Features
- Kalray SigmaC Language Toolchain
- Related Dataflow Languages and Toolchains
- Kalray MPPA® Application Examples
- Conclusions
Lessons Learned

- Kalray Dataflow well suited to key MPPA® applications
  - Cyclostatic dataflow especially effective on signal processing and video encoding (AVC/H264, HEVC/H265)
  - Other applications that deploy data-dependent computation graphs (such as LTE base station) are more difficult to express

- Static Dataflow allows to automate parallel execution on clustered manycore processors such as the MPPA®-256
  - Code and data distribution, communication over NoC
  - No need for specific architectural support in NoC and DMA

- Kalray Dataflow toolchain also enables parallel execution of single applications on hybrid target systems
  - Demonstrated Intel CPU + 2 MPPA® AB01 boards
Future Developments

- Extended Cyclostatic Dataflow Techniques
  - Based on work by A. Munier et O. Marchetti (U. Paris VI / LIP6) on Marked Weighted Timed Event Graphs (MTWEG)
  - K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph
    B. Bodin, A. Munier-Kordon, B. Dupont de Dinechin
    Embedded Computer Systems (SAMOS), 2012
  - Liveness evaluation of a cyclo-static DataFlow graph
    M. Benazouz, A. Munier-Kordon, T. Hujsa, B. Bodin
    Proceedings of the 50th Annual Design Automation Conference
  - Periodic Schedules for Cyclo-Static Dataflow
    Accepted at ESTIMedia 2013

- Time-Triggered source and sink nodes, RT extensions