A Distributed Run-Time Environment for the Kalray MPPA®-256 Integrated Manycore Processor

Benoît Dupont de Dinechin

ICCS 2013 ALCHEMY
June 5th 2013
KALRAY, a global solution

Powerful and Programmable Processors
- 256 VLIW cores – 28nm technology
- High performance @ low power consumption
- Fully software programmable

Software Development Kit (SDK)
- Manycore compiler, simulator, debugger, profiler, GUI
- Programming from high-level C-based language

Development platform
- “Ready to use” platform
- Hardware and Software solution for application development
- Access to full computing power of the MPPA® processors
First MPPA®-256 Chips with CMOS 28nm TSMC

- High processing performance
  700 GOPS – 230 GFLOPS
- Low power consumption
  5W – 10W
- High execution predictability
- Sampled November 2012

Real-time H264 encoder running under 6W on MPPA®-256
MPPA® DEVELOPER Highlight

- Develop, optimize and evaluate your applications
- Access the full computing power of the 256 cores
- “Ready to develop” concept (no specific set-up)

- PCIe board MPPA®-256 Processor
- PCIe board for debug/probe
- Intel core I7 CPU 3.6GHz, Linux OS
- MPPA ACCESSCORE SDK installed
- Compatible with Multi MPPA board
- Additional services:
  - Extranet access
  - Support Team access
  - Getting started training
  - SDK maintenance
MPPA®-256 Processor Hierarchical Architecture

VLIW Core

Compute Cluster

Manycore Processor

Instruction Level Parallelism

Thread Level Parallelism

Process Level Parallelism
MPPA®-256 Distributed Memory Architecture

- 20 memory address spaces
  - 16 compute clusters
  - 4 I/O subsystems with DDR access
- Dual Network-on-Chip (NoC)
  - Data NoC & Control NoC
  - Full duplex links, 32 bits / cycle
  - 2D torus topology + extension links + jumper links
- NoC with wormhole switching
  - Message is a sequence of packets
  - Packets are atomically received
  - Each packet has a header and a payload, composed of flits
  - The header flit(s) contains the route, port, and other information
MPPA®-256 NoC Configuration

- **Data NoC Tx interface**
  - Set route to destination, port, unicast/multicast
  - Optionally set notification bit in packet header
  - Optionally set offset-based Rx mode (default to sequential)
  - Optionally set QoS parameters

- **Data NoC Rx interface**
  - Port associated with a Rx buffer in memory
  - RM core event on notification count or data byte count

- **Control NoC Tx interface**
  - Set route to destination, port, unicast/multicast
  - Optionally set notification bit in packet header

- **Control NoC Rx interface**
  - Port associated with a 64-bit mailbox
  - RM core event on single or OR-ed notification
**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
  - Restrictive CUDA or OpenCL 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® adapts the supercomputer distributed memory model**
  - Distributed memory architecture scales to > 1M cores (BlueGene/Q)
  - Enables low energy per operation and high execution predictability
Kalray Software Development Kit
MPPA ACCESSCORE – MPPA ACCESSLIB

Standard C/C++ Programming Environment

Simulators, Debuggers & System Trace

Operating Systems & Device Drivers

Dataflow Programming
FPGA Style

POSIX-Level Programming
DSP Style

OpenCL Programming
GPU Style

Today
Q4 2013
POSIX-Level Programming Environment

- POSIX-like process management
  - Spawn 16 processes from the I/O subsystem
  - Process execution on the 16 clusters start with main(argc, argv) and environment
- Inter Process Communication (IPC)
  - POSIX file descriptor operations on ‘NoC Connectors’
  - Extension to the PCIe interface with the ‘PCI Connectors’
  - Rich communication and synchronization
- Multi-threading inside clusters
  - Standard GCC/G++ OpenMP support
    - #pragma for thread-level parallelism
    - Compiler automatically creates threads
  - POSIX threads interface
    - Explicit thread-level parallelism
POSIX IPC with NoC Connectors

- Build on the ‘pipe & filters’ software component model
  - Processes are the atomic software components
  - NoC objects operated through file descriptors are the connectors:

<table>
<thead>
<tr>
<th>Connector</th>
<th>Purpose</th>
<th>Tx:Rx Endpoints</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sync</td>
<td>Half synchronization barrier</td>
<td>N:1, N:M (multicast)</td>
<td>CNoC</td>
</tr>
<tr>
<td>Portal</td>
<td>Remote memory window</td>
<td>N:1, N:M (multicast)</td>
<td>DNoC</td>
</tr>
<tr>
<td>Stream</td>
<td>Remote circular buffer</td>
<td>1:1, 1:M (multicast)</td>
<td>DNoC</td>
</tr>
<tr>
<td>RQueue</td>
<td>Remote atomic enqueue</td>
<td>N:1</td>
<td>DNoC+CNoC</td>
</tr>
<tr>
<td>Channel</td>
<td>Zero-copy rendez-vous</td>
<td>1:1</td>
<td>DNoC+CNoC</td>
</tr>
</tbody>
</table>

- Synchronous operations: open(), close(), ioctl(), read(), write(), pwrite()
- Asynchronous I/O operations on Portal, Stream, Queue
  - Based on aio_read(), aio_error(), aio_return()
  - NoC Tx DMA engine activated by aio_write()
Data Types for Asynchronous POSIX Operations

- **Struct sigevent**

  ```c
  struct sigevent {
    int sigev_notify; /* Notification type: SIGEV_NONE, SIGEV_CALLBACK */
    union sigval sigev_value; /* Signal value : pointer to enclosing struct aiocb */
    void (*sigev_notify_function)(union mppa_sigval); /* Notification function */
  };
  ```

- **Struct aiocb**

  ```c
  struct aiocb {
    int aio_fildes; /* File descriptor */
    int aio_lio_opcode; /* If <= 0 then read else write */
    ssize_t aio_offset; /* File offset */
    volatile void *aio_buf; /* Location of buffer */
    size_t aio_nbytes; /* Length of transfer */
    struct sigevent aio_sigevent; /* Signal number and value */
  };
  ```
Extensions to POSIX Asynchronous Operations

- `mppa_aio_wait(aiocb)`
  - Sequence of `aio_suspend(&aiocb, 1, 0), aio_return(aiocb)`
- `mppa_aio_rearm(aiocb)`
  - Atomic sequence of `aio_wait(aiocb), aio_read(aiocb)`
- `mppa_pwrites(fd, base, esize, ecount, sstride, tstride, offset)`

- `mppa_aiocb_set_pwrite(aiocb, base, size, offset)`
- `mppa_aiocb_set_strides(aiocb, ecount, sstride, tstride)`
POSIX-Level for Acceleration over PCI

- MPPA® process management from the host
  - Load/unload MPPA® multi-binary file: mppa_load(), mppa_unload()
  - POSIX-level process management: mppa_spawn(), mppa_waitpid()
  - POSIX-level IPC through PCI connectors:

<table>
<thead>
<tr>
<th>Connector</th>
<th>Purpose</th>
<th>Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Memory buffer in Rx process</td>
<td>PCI DMA, DDR</td>
</tr>
<tr>
<td>MQueue</td>
<td>POSIX-like message queue</td>
<td>I/O subsystem SMEM</td>
</tr>
</tbody>
</table>

- Buffer connector operations
  - Tx process: aiocb_set_pwrite(), aiocb_set_strides(), aio_write()
  - Rx process: aiocb_set_trigger(), aio_read()
  - Local completion on Rx or Tx process: aio_wait()
  - No remote completion
POSIX-Level & Distributed Computing

- Split-phase barrier (master-slave implementation)
  - Arrival phase by using the Sync connector in N:1 mode
  - Departure phase by using the Sync connector in 1:M mode

- Active message server
  - Use RQueue connector with asynchronous call-back on aio_read()
  - Call-back function executes on the RM and must rearm aio_read()

- Split-C put() and get() remote memory operations
  - Remote put() mapped to Portal connector operations
  - Remote get() implemented by active messages that operate a Portal

- Rendez-vous
  - Channel connector with synchronous read() and write()
MPPA® in Audio Application: a multi DSP + control on a chip

*Increase performances and Reduce Audio system cost*

- Multi Channel processing
  - 256 VLIW cores ~ 500 Low End DSPs
- Channel routing and control
  - Dual NoC + 32 integrated DMAs
- System integration
  - Up to 8 x Ethernet 1GbE
- Low Latency audio processing
  - 500µs latency from input to output samples
- Cost effective
  - Equivalent to complex multi DSPs + FPGAs system
MPPA® in Computational Finance

- Monte Carlo method
- Optimized pseudo random generator
- Parallel Map / Reduce scheme scaling on multiple MPPA
- Optimized mathematical primitives for Kalray core
- Real test case: foreign exchange option pricing

Power efficiency 5x better than recent GPU
MPPA® on Dense Matrix Multiplication

- 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

1: Let \( q = \text{pid} \)
2: Let \( p_i = q \mod \sqrt{p} \)
3: Let \( p_j = q/\sqrt{p} \)
4: Let \( C_q = 0 \)
5: for \( 0 \leq l < \sqrt{p} \) do
6: \( a \leftarrow A_{((p_i+p_j+l) \mod \sqrt{p})\sqrt{p}+p_i} \)
7: \( b \leftarrow B_{((p_i+p_j+l) \mod \sqrt{p})+\sqrt{p}\times p_j} \)
8: sync
9: Let \( b^t = \text{transpose}(b) \)
10: \( C_q += a \times^t b^t \)
11: end for

\[ \sqrt{p} \]

\[ \sqrt{p} \]

"get from matrix A"

"get from matrix B"
Summary and Conclusions

- Features of the POSIX-Level programming model
  - Full access to MPPA execution resources using POSIX concepts:
    - One process per compute cluster
    - NoC and PCIe Inter-Process Communication
    - One POSIX or OpenMP thread per user core
    - Require that code generation process manage memory distribution
      - Maximum of 2MB of code + data in each compute cluster
      - Explicit data transfers from/to the DDR memory
    - Direct access to the MPPA performances and timing predictability
  - Evolutions of the POSIX-Level programming model
    - Dynamic code loading => atomic components are ‘load modules’
    - Execution partitions inside a MPPA or across several MPPA chips
    - Support of macro-operations such as MPI collectives