A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications

Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoît Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, Thierry Strudel

Kalray SA, 445 rue Lavoisier, F-38330 Montbonnot, France

***Abstract*—The Kalray MPPA-256 processor integrates 256 user cores and 32 system cores on a chip with 28nm CMOS technology. Each core implements a 32-bit 5-issue VLIW architecture. These cores are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them is ensured by data and control Networks-On-Chip (NoC). The MPPA-256 processor is also fitted with a variety of I/O controllers, in particular DDR, PCI, Ethernet, Interlaken and GPIO.**

We demonstrate that the MPPA-256 processor clustered manycore architecture is effective on two different classes of applications: embedded computing, with the implementation of a professional H.264 video encoder that runs in real-time at low power; and high-performance computing, with the acceleration of a financial option pricing application. In the first case, a cyclostatic dataflow programming environment is utilized, that automates application distribution over the execution resources. In the second case, an explicit parallel programming model based on POSIX processes, threads, and NoC-specific IPC is used.

Introduction

The Kalray MPPA-256 is a single-chip manycore processor manufactured in 28nm CMOS technology that targets low to medium volume professional applications, where low energy per operation and time predictability are the primary requirements. Its 256 user cores and 32 system cores are distributed across 16 compute clusters of 16+1 cores, and 4 quad-core I/O subsystems. Each compute cluster and I/O subsystem owns a private address space, while communication and synchronization between them is ensured by data and control Networks-on-Chip (NoC).

The software development kit provides standard GNU C/C++ & GDB development tools for compilation & debugging at the cluster level, SMP Linux running on the I/O subsystems, and a lightweight POSIX kernel running on the compute clusters. Based on these foundations, two programming models are currently supported:

- A cyclostatic dataflow language based on C syntax, whose compiler automatically sizes the communication buffers and distributes the tasks across the memory spaces and the cores.
- POSIX-Level, where POSIX processes are spawned to execute on the compute clusters. Inside processes, standard POSIX threads and OpenMP are available for multi-threading across the cores. Specific IPC takes advantage of the NoC architecture.

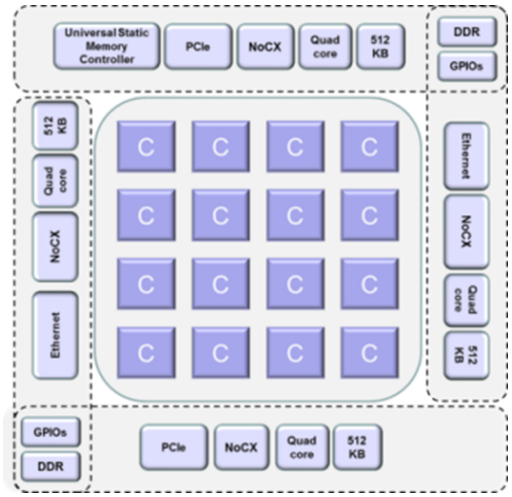

Fig. 1. MPPA manycore architecture.

These two programming models are available whether the MPPA-256 runs in a stand-alone configuration, or as an accelerator connected via a PCIe link to a host CPU. In case of the accelerator configuration, the two programming models manage the distribution of application code and abstract communications between the host CPU and the MPPA-256.

I. The MPPA-256 Processor

A. Manycore Architecture

The MPPA-256 chip integrates 16 compute clusters and 4 I/O subsystems located at the periphery of the processing array (Figure 1) to control all the I/O devices. Each I/O subsystem contains a SMP quad-core with a shared D-cache, on-chip memory, and a DDR controller for access to up to 64GB of external DDR3-1600. They run either a rich OS such as Linux or a RTOS that supports the MPPA I/O device drivers. The I/O subsystems integrate controllers for a 8-lane Gen3 PCI Express for a total peak throughput of 16 GB/s full duplex, Ethernet links ranging from 10Mb/s to 40Gb/s for a total aggregate throughput of 80Gb/s, Interlaken link providing a way to extend the NoC across MPPA-256 chips, and other I/O devices in various configurations like UARTs, I2C, SPI, Pulse Width Modulator (PWM) or General Purpose IOs (GPIOs).

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

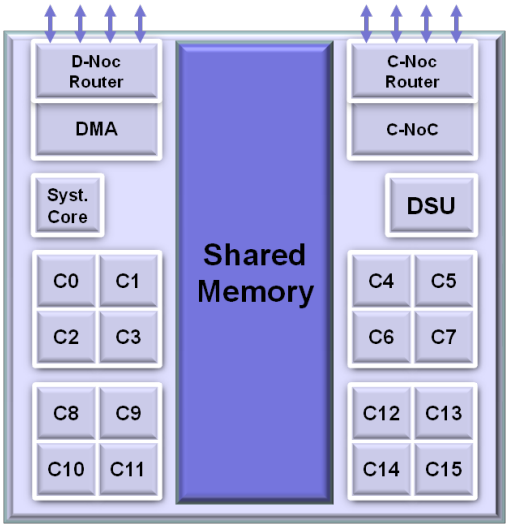

Fig. 2. MPPA compute cluster.

The 16 compute clusters and the 4 I/O subsystems are connected by two explicitly addressed NoC with bi-directional links, one for data (D-NoC), and the other for control (C-NoC). The two NoC are identical with respect to the nodes, the 2D torus topology, and the wormhole route encoding. They differ at their device interfaces, by the amount of packet buffering in routers, and by the flow regulation at the source available on the D-NoC. NoC traffic through a router does not interfere with the memory buses of the underlying I/O subsystem or compute cluster, unless the NoC node is a destination.

B. Compute Clusters

The compute cluster (Figure 2) is the basic processing unit of the MPPA architecture. Each cluster contains 16 processing engine (PE) cores, one resource management (RM) core, a shared memory, a direct memory access (DMA) engine responsible for transferring data between the shared memory and the NoC or within the shared memory. The DMA engine supports multi-dimensional data transfers and sustains a total throughput of 3.2GB/s in full duplex.

The compute cluster shared memory architecture optimizes a trade-off between area, power, bandwidth, and latency. The shared memory totals 2MB, organized in 16 parallel banks of 128KB each, with Error Code Correction (ECC) on 64-bit words. The shared memory delivers an aggregate bandwidth of 38.4 GB/s. Each bank arbitrates between twelve master ports and guarantees simultaneous accesses to each client in steady state. These master ports comprise: 8 ports for the 16 PE cores after pre-arbitration at core pairs, one port for the RM core, two ports for the data NoC, and one port the DSU.

The Debug & System Unit (DSU) supports the compute cluster debug and diagnostics capabilities. Each DSU is connected to the outside world by a JTAG (IEEE 1149.1) chain. The DSU also contains a system trace IP that is used by lightly instrumented code to push up to 1.6Gb/s of trace data to an external acquisition device. This trace data gives live and almost non-intrusive insight on the behavior of the application.

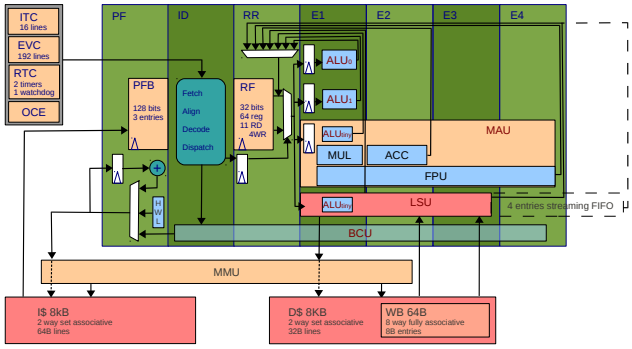

Fig. 3. VLIW core pipeline.

C. VLIW Core

Each PE or RM core (Figure 3) implements a 5-way VLIW architecture with two arithmetic and logic units, a multiply-accumulate & floating-point unit, a load/store unit, and a branch & control unit, enabling up to 2 GOPS or 800 MFLOPS on 32-bit data at 400MHz. These five execution units are primarily connected through a shared register file of 64 32-bit general-purpose registers (GPRs), which allows for 11 reads and 4 writes per cycle.

Each core sees the memory through a level-1 cache. Separate 2-way associative instruction and data caches of 8KB each are provided. Cache coherence is enforced by software, although this is not visible from applications that rely on the proper use of OpenMP directives or POSIX threads synchronizations. Each core also features a MMU for process isolation and virtual memory support.

II. Dataflow Language and Toolchain

A. Why Another Language?

The cyclostatic dataflow (CSDF) model of computation [1] is a specialization of Kahn process networks (KPN) [2], which presents ideal properties for efficiently scheduling and mapping applications on manycore architectures:

- Programmers do not need to care about the synchronization between computations. Only data dependencies are expressed and computing *agents* are activated only after all their dependencies are satisfied.
- Computing *agent*s are only connected through well identified *channels*. No mutable shared variables, global mutable variables or side effects between *agent*s are possible in this model.

In CSDF, the effects of computing on channels is modeled by an independent cyclic finite state machine triggered by each *agent* activation. This means that the application behavior is predictable and independent from data processing.

Kalray provides a C based CSDF programming language named ΣC [3]. ΣC has been designed to allow programmers to reuse existing C code, while benefiting from the advantages of the dataflow model [3]. Furthermore, the ΣC compilation toolchain adapts as far as possible the generated code to the

target (number of cores, memory available, NoC topology, ...). The use of genericity, by the means of instance parameters, allows parametric dataflow graph description. A simple change of the parameters followed by a recompilation adapts the level of parallelism to different performance or resource constraints.

Listing 1 presents code that reads a character stream, reverses the characters for a given length, and then outputs the result as another character stream. An *agent* ('reverser' in this example) can be seen as an autonomous executable component which reads data, processes it and writes it.

```
agent reverser(int size) {
  interface {
    in<char> input_port;
    out<char> output_port;
    spec {input_port[size];
    output_port[size]};
  }
  void start (void) exchange
    (input_port input[size],
    output_port output[size]){
    for (int i = 0; i < size; ++i)
      output[i] = input[size-i];
  }
}
```

Listing 1. A simple agent that reverse its input.

B. Compilation Overview

The ΣC compilation is decomposed into 3 distinct phases:

- Instantiation: ΣC code is converted to C code, compiled and executed to generate a representation of the dataflow graph, its elements, channels and automata.
- Mapping: Using the generated description, the application is scheduled, placed and routed on the target platform according to both hardware restrictions and user performance constraints. A run-time is generated for each binary.
- Compilation: Using the converted source files and the generated run-time, the application is compiled with our lightweight system software.

The key feature of the ΣC toolchain is that although the language is completely platform agnostic, the compilation process can and will automatically distribute a parallel application across a large number of processors, using both different architectures and communication links. This essentially means that the same parallel ΣC application can be executed on a native x86 processor, as well as on both a x86 and a MPPA using PCI and NoC interfaces, but also be scaled to multiple hosts and multiple MPPA chips.

The ΣC compiler also provides a memory-saving feature called 'agent inlining'. In almost every dataflow application, it is required to either duplicate, scatter or gather the data to efficiently parallelize the application. While supporting such patterns, the ΣC compiler goes a step further and provides an interface for describing agents with complex data movement patterns. Through analysis of the surrounding automata and with symbolic execution, the ΣC compiler is able to 'inline' such an agent into a shared buffer, that can be safely accessed by both inputs and outputs, or into a micro-coded task to be executed by a dedicated DMA engine.

C. Mapping an Application

The ΣC mapping tool, `sc-map`, is the core of the toolchain. Using a customizable platform description, the dataflow graph, and user constraints, the mapping tool solves three problems:

a) Scheduling: During the scheduling, `sc-map` solves three issues. It analyzes all agents to determine which agents could be inlined, thus reducing the memory footprint, it sizes all communication channels to ensure the liveness of the application, and computes the minimum pipelining required to achieve the required performance [4]. Altogether, it guarantees the proper execution of the application whatever platform it may be executed on.

b) Mapping: Using heuristics, `sc-map` maps the dataflow graph on the targeted platform which can be anything from a single x86 CPU to a multi MPPA board. The algorithm's goal is to find a mapping that respects hardware restrictions (memory space, network bandwidth, CPU time) while minimizing the impact of network communications. The ΣC toolchain also provides a GUI to interact with the placer, by either by helping it or by providing a custom mapping.

c) Routing: As the MPPA D-NoC offers QoS, `sc-map` computes and allocates routes in order to limit network latencies and guarantee the required bandwidth.

D. System Software

Finally, the ΣC toolchain provides a lightweight System Software to provide run-time support for ΣC application. For under 25KB of memory footprint, the ΣC System Software provides a run-time support for ΣC applications to run on POSIX platforms or custom operating systems (as in compute clusters), with communication support. With context switches under 1μs in compute clusters, its low overhead permits both high performance and small task granularity.

III. Application to H.264 Encoder

A. H.264 Encoder Overview

H.264/MPEG-4 Part 10 or AVC (Advanced Video Coding) is a standard for video compression, and is currently one of the most commonly used formats for the recording, compression, and distribution of high definition video.

High quality H.264 video encoding requires high compute power and flexibility to handle the different decoding platforms, the numerous image formats, and the various application evolutions. On the other hand, video encoding algorithms exhibit large parallelism suitable for efficient execution on manycore processors. This kind of application can then be developed using the ΣC environment in order to describe task parallelism when addressing manycore architectures, such as the MPPA processor.

B. H.264 Encoder Description Using ΣC

The H.264 encoding process consists in separately encoding many macro-blocks from different rows. This is the first level of parallelization, allowing a scalable encoding application, where a various number of macro-blocks can

be encoded in parallel. In this graph, each "Encode MB Process" sub-graph exploits this data parallelism. Fine grained task parallelism is also described: motion estimation on each macro-block partition (up to 4x4), spatial prediction of intra-coded macro-blocks, RDO analysis and trellis quantization are performed concurrently in separate agents.

The ΣC compiler analyzes the dataflow graph and gives to the user an overview of the scheduling of the application, using profiling data. It is also able to map the application onto the targeted MPPA architecture, and implements all communication tasks between ΣC agents.

C. Challenges of Cyclostatic Dataflow Descriptions

The ΣC environment supports cyclostatic dataflow application, with execution based on a steady state. The application then exchanges defined amounts of data, independent of run-time states or incoming data: in the H.264 algorithm, the amount of data differs according to image type (intra or inter), but the ΣC application always works with data for both cases.

Describing and managing search windows for motion estimation is another challenge when using a dataflow environment: difficulties describing delay and shared memory between different function blocks. Fortunately, the ΣC environment implements different kinds of features (including virtual buffers and delays) allowing an efficient implementation (no unnecessary copies, automatic management of data, etc.)

D. Results and Performance

From the current implementation of the H.264 encoder using ΣC, performance analysis has been performed to determine the encoder global quality. These performance results have been compared to the initial x264 library, applied on different video sequences frequently used for such analysis.

This analysis leads to several conclusions listed below:

- From quality analysis based on bit-stream size and decoded video quality (using SSIM and PSNR criteria), the parallelized H.264 application using ΣC dataflow language offers better results than the initial x264 library. Using the MPPA manycore architecture leads to a freer implementation (fewer thresholds, less bypass, etc.). For example, many motion vectors can be tested in parallel, as well as many intra predictors, without impacting encoder speed. Finally, much more information is available, enabling a better solution, impacting the resulting encoder quality.
- Implementation of the x264 library on MPPA processor offers a real-time encoder for embedded solutions, and low-power needs. Results on a 720p video are:
  - Intra I-frame: about 110 frames per second
  - Inter P-frame: about 40 frames per second
  - Inter B-frame: about 55 frames per second
- Using a configuration equivalent to the implementation on MPPA, the x264 encoder has been executed on an Intel Core i7-3820 (4 hyper-threaded cores). All CPU capabilities have been used, including MMX2, SSE2Fast, SSE3, fastShuffle and SSE4.2. Resulting average performance is about 50 fps for both processors (Intel Core i7 and MPPA-256). Below are performance comparisons:

Processor	Performance	Power efficiency
Intel Core i7-3820	49 fps	2,60 W/fps
Kalray MPPA-256	52 fps	0,14 W/fps

For same H.264 encoding performance, the Kalray MPPA-256 processor is more power efficient by a factor of about 20, with a measured power consumption slightly above 5W.

IV. POSIX-Level Programming

A. Design Principles

The principle of MPPA POSIX-level programming is that processes on the I/O subsystems spawn sub-processes on the compute clusters and pass arguments through the traditional `argc`, `argv`, and `environ` variables. Inside compute clusters, classic shared memory programming models such as POSIX threads or the OpenMP language extensions supported by GCC must be used to exploit more than one core. The main difference between the MPPA POSIX-level programming and classic POSIX programming appears on inter-process communication (IPC). Precisely, IPC is only achieved by operating on special files, whose pathname is structured by a naming convention that fully identifies the NoC resources used when opening either in read or write mode (Table I).

This design leverages the canonical 'pipe-and-filters' software component model [5], where POSIX processes are the atomic components, and communication objects accessible through file descriptors are the connectors [6]. Like POSIX pipes, those connectors have distinguished transmit (Tx) and receive (Rx) ports that must be opened in modes `O_WRONLY` and `O_RDONLY` respectively. Unlike pipes however, they may have multiple Tx or Rx endpoints, and support POSIX asynchronous I/O operations with call-back. Following the components and connectors design philosophy, the NoC node or node sets at the endpoints are completely specified by the connector pathnames.

B. NoC Connectors Summary

Sync A 64-bit word in the Rx process that can be OR-ed by N Tx processes. When the result of the OR equals -1, the Rx process is notified so a `read()` returns non-zero.

Signal A 64-bit message is sent by a Tx process to M Rx processes. Each Rx process is notified of the message arrival which is typically handled by an `aio_read()` call-back.

Portal A memory area of the Rx process where N Tx processes can write at arbitrary offsets. The Rx process is not aware of the communication except for a notification count that unlocks the Rx process when the `trigger` supplied to `aio_read()` is reached.

Stream Message broadcast from one Tx process to several Rx processes. A Rx process is not aware of the communication but is ensured to find a valid and stable pointer to the latest message sent. This connector is provided to implement the communication by sampling (CbS) mechanism [7].

Type	Pathname	Tx:Rx	`aio_sigevent.sigev_notify`
Sync	`/mppa/sync/`*rx_nodes*`:`*cnoc_tag*	$N:M$	
Signal	`/mppa/signal/`*rx_nodes*`:`*cnoc_tag*	$1:M$	`SIGEV_NONE`, `SIGEV_CALLBACK`
Portal	`/mppa/portal/`*rx_nodes*`:`*dnoc_tag*	$N:M$	`SIGEV_NONE`, `SIGEV_CALLBACK`
Stream	`/mppa/stream/`*rx_nodes*`:`*dnoc_tag*	$1:M$	`SIGEV_NONE`, `SIGEV_CALLBACK`
RQueue	`/mppa/rqueue/`*rx_node*`:`*dnoc_tag*`/`*tx_nodes*`:`*cnoc_tag*`/`*msize*	$N:1$	`SIGEV_NONE`, `SIGEV_CALLBACK`
Channel	`/mppa/channel/`*rx_node*`:`*dnoc_tag*`/`*tx_node*`:`*cnoc_tag*	$1:1$	`SIGEV_NONE`, `SIGEV_CALLBACK`

TABLE I. NoC CONNECTOR PATHNAMES, SIGNATURE, AND ASYNCHRONOUS I/O SIGEVENT NOTIFY ACTIONS.

RQueue Atomic enqueue of *msize*-byte messages from several Tx processes, and dequeue from a single Rx process. The RQueue connector implements the remote queue [8], with the addition of flow control. This is an effective $N:1$ synchronization primitive [9], by the atomicity of the enqueue operation [10].

Channel A communication and synchronization object with two endpoints. Default behavior is to effectuate a rendez-vous between the Tx process and the Rx process, which transfers the minimum of the sizes requested by the read and the write without intermediate copies.

C. Support of Distributed Computing

Split Phase Barrier The arrival phase of a master-slave barrier [11] is directly supported by the Sync connector, by mapping each process to a bit position. The departure phase of a master-slave barrier [11] is realized by another Sync connector in $1:M$ multi-casting mode.

Active Message Server Active messages integrate communication and computation by executing user-level handlers which consume the message as arguments [12]. Active message servers are efficiently built on top of remote queues [8]. In case of the RQueue connector, the registration of an asynchronous read user call-back enables to operate it as an active message server.

Remote Memory Accesses One-sided remote memory access operations (RMA) are traditionally named PUT and GET [12], where the former writes to remote memory, and the latter reads from remote memory. The Portal connector directly supports the PUT operation on a Tx process by writing to a remote D-NoC Rx buffer in offset mode. The GET operation is implemented by active messages that write to a Portal whose Rx process is the sender of the active message.

V. APPLICATION TO OPTION PRICING

A. Monte Carlo Option Pricing with MPPA

A common use case of compute farms in the financial market is option pricing evaluation. Monte Carlo are widely used methods for option pricing option. They require the evaluation of tens of thousands of paths, each one divided in hundreds of computational steps. A simple pricing option evaluation requires a data set of hundreds of megabytes. Each path can be computed independently, thus the application exhibits a high level of parallelism and is a good candidate for offloading to an accelerator.

We implemented on the MPPA a typical option pricing application using a Monte Carlo method. This implementation had to meet two requirements: firstly, provide a simple acceleration interface that can be targeted from a customer native environment; secondly, distribute data and computation kernels efficiently across the MPPA compute clusters.

B. Acceleration interface

In order to ease the access to a MPPA accelerator on a classical customer framework, a Parallel-For Parallel-Reduce programming pattern has been used. The option pricing evaluation is performed by computation kernels. A kernel is a Plain Old Data (POD) structure that defines an operator. Therefore, they can be easily manipulated and copied from/to the accelerator. These kernels are iterated over each path and thus process the whole data set.

The Parallel-For interface allows to explicitly parallelize the computation over the provided data set. In this case, the parallel dimension is the tens of thousands of Monte Carlo paths. In the end, the result reduction is also performed in parallel by the Parallel-Reduce invocation.

In order to execute the above programming model efficiently on the MPPA, a run-time has been developed making use of the POSIX-level programming interface with the Portal, Sync and RQueue connectors. The connector semantics perfectly matches the pipelined computation model and allows the programmers to concentrate their effort on the use-case.

C. Kernels and data set distribution

Due to the nature of algorithm, the data set pattern access exhibits poor temporal locality. Most of the input data set is used once and never accessed again. Consequently, caching data would not provide any benefits.

For implementing this use-case on the MPPA we opted for a pipelined computation structure. The data set is organized into independent slices. Then, a computation pipeline is built between the IO subsystems and the compute clusters. The IO subsystems fill-in the compute clusters with batches of data set slices. The data transfers are overlapped with the computation of the previous slices on the clusters.

There is little resident data in the cluster's memory: the compute kernel code and associated structures, temporary variables and constant data. The resident data set is replicated on all the clusters. In particular, kernels and constant data.

Each data slice exhibits strong spatial locality. Detailed execution profiling shows that processing on the compute clusters does not suffer from data access latencies. The execution efficiency is nearly optimal on each core.

D. MPPA key features

This use-case takes advantage of unique features provided by the MPPA architecture. Some of them are:

- NoC broadcast capability: heavily used to download the compute kernels and constant data on all the compute clusters in parallel.
- DMA engine programmable threads: used to build on-the-fly dense data set slices from the main data set stored in DDR memory.
- NoC bidirectional links: allow to upload and download data in parallel with compute cluster processing without compromising the performance.
- VLIW architecture and instruction bundle structure: kernel codes fully exploit these features to speedup the computation.

E. Execution results

The accelerated part of the application is executed on the MPPA-256 400MHz processor, which is rated at 58 GFLOPS DP peak for 15W. This accelerated part is also executed on the two customer production platforms: the Intel i7-3820 quad-core 3.60GHz CPU rated at 115.2 GFLOPS DP peak for 130W when the 8 logical cores are exploited; and the NVIDIA Tesla C2075 GPU rated at 515 GFLOPS DP peak for 225W.

Accelerator	Time (s)	Performance	Energy (J)
i7-3820	13.86	0.17	1802.2
Tesla C2075	2.37	1.00	531.7
MPPA-256	5.75	0.41	86.3

Although the MPPA-256 has a significantly lower peak performance than the CPU and the GPU for double precision floating-point arithmetic, on this application its outperforms the CPU by a factor of 2.4× and delivers almost half the performance of the GPU. When comparing the energy consumed, the MPPA-256 is over 20 times more efficient than the CPU and over 6 times more efficient than the GPU.

CONCLUSIONS

The Kalray MPPA-256 manycore processor embeds 256 user cores and 32 system cores on a single 28nm CMOS chip running at 400MHz. These cores implement a modern 32-bit 5-issue VLIW architecture, with a floating-point unit and a memory management unit. This manycore processor is the first commercial product that integrates on a single chip a clustered architecture similar to high-end supercomputers. Unlike the shared memory architecture implemented by other manycore processors, a distributed memory architecture has no scalability limits, and enables highly energy efficient implementations.

Our results demonstrate that the MPPA clustered architecture can be successfully exploited on different applications. Specifically, we take advantage of a cyclostatic model of computation to fully automate the distribution of an embedded application across the memory, processing, and communication resources of the MPPA-256 manycore processor. We also show that programming with explicit management of these resources under familiar POSIX abstractions is effective for an accelerated computing application. In all cases, we observe high performances and over 20 times better energy efficiency compared to a high-end CPU.

On-going work focuses on the support of the OpenCL task parallel programming model, whose main advantage over the two programming models discussed is to enable direct access from any core to the external DDR memory. Our OpenCL support leverages the MMU available on each core to map the memory pages to the compute cluster memory, that is, it implements a software distributed shared memory. The key issue is to reconcile changes to pages at the same virtual addresses by different clusters. This will be addressed by adapting the TreadMarks [13] false sharing resolution technique.

REFERENCES

[1] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cycle-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, 1996.

[2] G. Kahn, "The semantics of a simple language for parallel programming," in *Proceedings of the IFIP Congress 74*, 1974, pp. 471–475.

[3] T. Goubier, R. Sirdey, S. Louise, and V. David, "ΣC: a programming model and langage for embedded many-cores," *LNCS*, no. 7016, pp. 385–394, 2011.

[4] B. Bodin, A. M. Kordon, and B. D. de Dinechin, "K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS XII*, 2012.

[5] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007.

[6] S. Kell, "Rethinking software connectors," in *International workshop on Synthesis and analysis of component connectors: in conjunction with the 6th ESEC/FSE joint meeting*, ser. SYANCO '07, 2007, pp. 1–12.

[7] A. Benveniste, A. Bouillard, and P. Caspi, "A unifying view of loosely time-triggered architectures," in *Proceedings of the tenth ACM international conference on Embedded Software*, ser. EMSOFT '10, 2010, pp. 189–198.

[8] E. A. Brewer, F. T. Chong, L. T. Liu, S. D. Sharma, and J. D. Kubiatowicz, "Remote queues: exposing message queues for optimization and atomicity," in *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '95, 1995, pp. 42–53.

[9] V. Papaefstathiou, D. N. Pnevmatikatos, M. Marazakis, G. Kalokairinos, A. Ioannou, M. Papamichael, S. G. Kavadias, G. Mihelogiannakis, and M. Katevenis, "Prototyping efficient interprocessor communication mechanisms," in *Proceedings of the 2007 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS 2007)*, 2007, pp. 26–33.

[10] M. G. Katevenis, E. P. Markatos, P. Vatsolaki, and C. Xanthaki, "The remote enqueue operation on networks of workstations," *International Journal of Computing and Informatics*, vol. 23, no. 1, pp. 29–39, 1999.

[11] O. Villa, G. Palermo, and C. Silvano, "Efficiency and scalability of barrier synchronization on noc based many-core architectures," in *Proceedings of the 2008 international conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '08, 2008, pp. 81–90.

[12] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation," in *Proceedings of the 19th annual International Symposium on Computer architecture*, ser. ISCA '92, 1992, pp. 256–266.

[13] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: distributed shared memory on standard workstations and operating systems," in *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, ser. WTEC'94. Berkeley, CA, USA: USENIX Association, 1994, pp. 10–10.