A Tightly-Coupled Multi-Core Cluster with Shared-Memory HW Accelerators

Masoud Dehyadegari*, Andrea Marongiu†, Mohammad Reza Kakoee†,
Luca Benini†,Siamak Mohammadi*,Naser Yazdani*
*School of ECE - University of Tehran - North Kargar st, Tehran, Iran
{*dehyadegari, smohammadi, yazdani*}*@ut.ac.ir*
†DEIS - University of Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy
{*a.marongiu, m.kakoee, luca.benini* }*@unibo.it*

***Abstract*—Tightly coupling hardware accelerators with processors is a well-known approach for boosting the efficiency of MPSoC platforms. The key design challenges in this area are: (i) streamlining accelerator definition and instantiation and (ii) developing architectural templates and run-time techniques for minimizing the cost of communication and synchronization between processors and accelerators. In this paper we present an architecture featuring tightly-coupled processors and hardware processing units (HWPU), with zero-copy communication. We also provide a simple programming API, which simplifies the process of offloading jobs to HWPUs.**

I. INTRODUCTION

Although Moore's law continues to offer exponential increases in transistor count, stringent power budgets for embedded chips limit the practical exploitation of integration capabilities to the extent they are matched by an equivalent improvement in energy efficiency. Multi-processor system-on-chip (MPSoC) technology has been successfully adopted since 2005 to deliver effective transistor usage, and is currently in the *many-core* era. Some researchers have recently outlined that core-count scaling will soon hit a *utilization wall* [20]. The ever-increasing number of on-chip transistors will make it impossible to supply power to all of them at the same time. As a result, as early as next year the most advanced chips will need to keep 21 percent of their transistors switched off at any one time, the so called *dark silicon*.

Architectural heterogeneity is an effective solution to improve energy efficiency of SoC designs. General-purpose multi-core processors can be coupled with hardware functional units, whose specialized logic can achieve 10× to 100× better energy efficiency over key computation-intensive activities. Authors of [21] report 11× lower energy consumption for their heterogeneous computing tile as compared to today's most energy efficient designs, and show how this approach can also make efficient use of dark silicon.

Heterogeneous platforms can provide significant improvements, energy- and performance-wise. However, to allow for the main program execution to exploit accelerators in an effective manner typically a lot of coding effort is required, mostly because of the accelerator memory model. In traditional acceleration techniques two distinct memory spaces for processors and accelerators are considered. This obviously complicates efficient accelerator exploitation, because the programmer has to explicitly transfer data appropriately and to keep copies of the same datum consistent between different memory spaces. Moreover, this communication infrastructure implies frequent data movements from one memory space to another – which in some cases are conceptually unnecessary but enforced by the memory model – thus undermining acceleration benefits.

Shared memory is a convenient abstraction to simplify both problems and allow for efficient processor-to-accelerator communication. Representative examples of such a solution are nowadays appearing, like the AMD Fusion [18]. However, similar proposals often only provide the abstraction of a shared memory to programmers, while the underlying memory system still relies on separate memory spaces and implicit copies. As a result, while writing accelerated programs results easier, silent data transfers may still hinder the performance. To solve this problem it is necessary to design efficient processor-to-memory shared memory communication infrastructures.

In this paper we propose a tightly-coupled multi-core cluster architecture named *Megaleon*, where several homogeneous parallel cores are coupled to a number of dedicated hardware accelerators named HW processing units (HWPU). To interface main program execution with HW accelerators we designed a dedicated architecture template where HWPU and CPU communicate through shared memory. In particular, we propose an architecture where the HWPUs share this same L1 data memory through which processors also communicate, thus leveraging a zero-copy communication model. A first advantage of this approach is that no data inconsistencies may arise, since we disallow the coexistence of multiple copies of the same datum in the system, thus freeing programmers from the burden of maintaining copies coherent. Second, most of the data transfers typically required in traditional acceleration approaches are no longer necessary with this design.

We set up an integrated design flow which starting from the C application specification leverages HLS tools to automatically generate RTL models for the target HWPU. We present a custom *wrapper* that interfaces these HDL HWPU description to RTL models of the whole *Megaleon* platform (cores, memory, interconnection). We also provide a software API to simplify the usage of HWPUs from a C program. Finally, we present a set of results with two representative applications from the image processing domain, where we explore different parallelization and acceleration alternatives. Our results confirm the effectiveness of our proposal.

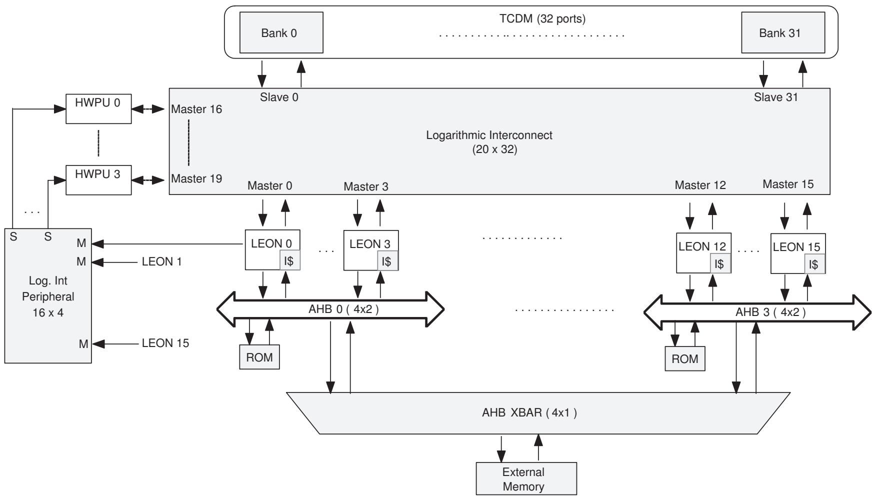


Fig. 1: Mega-leon architecture.

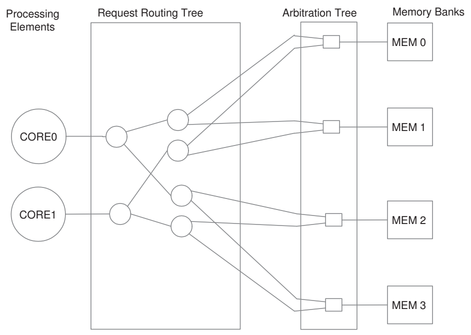


Fig. 2: Mesh of trees 2x4: empty circles represent routing switches and empty squares represent arbitration switches.

The rest of the paper is organized as follows: Section II surveys related work to ours. In Section III we present our tightly-coupled shared memory cluster, describing in details the communication interface (*wrapper*) between processors and HWPUs. In Section IV we assess and validate the effectiveness of our approach, while Section V summarizes our contribution and findings.

II. Related Work

Due to their high potential computing power, hardware accelerators such as graphical processing units (GPUs) are very effective to speed-up applications on multi-processor system on chips [1] [14]. These devices achieve high performance with highly parallel microarchitecture and fast internal memories [1]. However, their performance is strongly affected by the data communication overhead between CPU and GPUs [1] [2]. Several techniques are proposed in the literature trying to reduce this communication overhead. Al-Kiswany et.al in [5] describe StoreGPU, a distributed storage system that uses pinned, non-pageable memory on the host system to reduce the impact of data transfer. Gelado et al. [4] introduce an “Asymmetric Distributed Shared Memory” (ADSM) model that provides two types of memory updates (“Lazy” and “Rolling”) that determine when to move data on and off the GPU. Becchi et al. [5] implement a heterogeneous scheduler that takes memory transfer overhead into consideration, and chooses not to migrate data when the overhead is too great. All these techniques are based on optimizing the data transfer and not avoiding it, completely.

Shared memory architectures are proposed trying to completely remove the overhead of data communication between CPU and hardware accelerators [10] [9] [12]. OpenCL [12] limits the shared address to GPUs and it is a low-level programming style and need to get a deep knowledge of underlying architecture. Intel’s C for Heterogeneous Integration (CHI) programming environment [9] is a different approach to tightly integrate accelerators such as GPUs and general purpose CPU cores together based on the proposed EXOCHI [9] model. EXOCHI supports a shared virtual memory heterogeneous multi-threaded programming model with minimal OS intrusion. However, it needs page tables for synchronization. Similarly, our approach is based on shared memory between processors and accelerators. Moreover, we propose a fully automatic flow to generate and integrate hardware accelerators into the system using a commercial HLS tool. A shared-memory accelerator model is also proposed in the GreenDroid processor [21]. However, the conservation cores (c-cores) that comprise the GreenDroid system have different goals compared to our accelerators. Fundamentally, c-cores focus on reducing the energy of executing code, even if they only modestly improve the resulting execution time.

III. Architecture

In this section we describe the target tightly-coupled shared memory cluster, as well as the communication interface (i.e., the wrapper) used by HWPUs to directly access the TCDM. We also describe how using this wrapper we can easily instantiate new HWPUs by leveraging HLS tools, and how this shared-memory processor-to-HWPU communication model simplifies the process of writing accelerated applications.

A. Mega-Leon Architecture

The Mega-Leon architecture is an evolution of the Multi-core Leon developed by Gaisler [16]. Its simplified block diagram is shown in Figure 1. It contains sixteen SPARC-V8 processor cores, each featuring a private instruction cache, which communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). The TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a high-bandwidth Mesh-of-Trees (MoT). The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. A fully-synthesizable MoT network suitable for shared-L1 processor clusters has been proposed in [7], featuring single-cycle transfer from processor to memory and vice versa. Our design is based

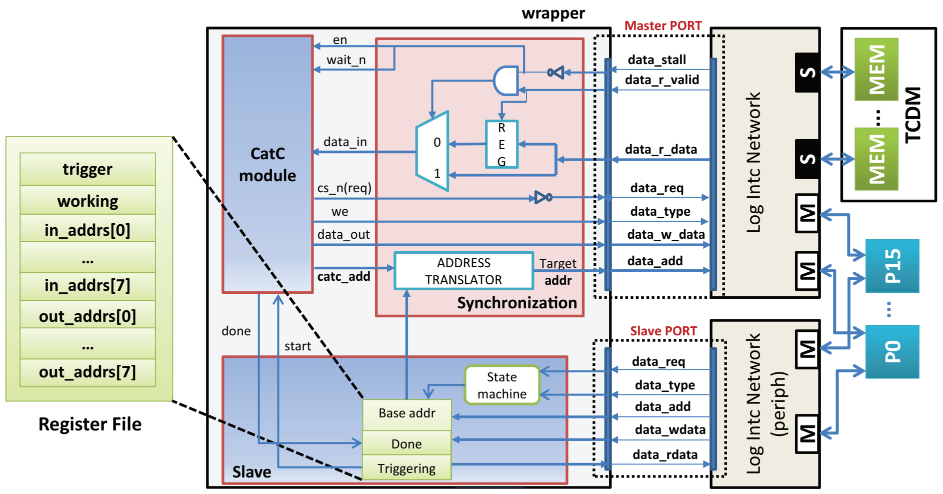


Fig. 3: Structure of HWPU

on this network: we use the architecture of this logarithmic interconnection to connect processors to TCDM. A MoT interconnection which connects 2 processors to 4 memory modules (2x4) is shown in Figure 2. The original interconnection [7], supports non-blocking communication between the processors and the memories modules, within a single clock. As shown in Figure 2, the MoT network connects N=2n processors and M=2m memories. It contains Log2M levels of routing primitives and Log2N levels of arbitration primitives. Each memory request issued by a processor must pass through Log2M levels of routing primitives to reach one of MxN leaf nodes in the arbitration switches. Moreover, it undergoes Log2N levels of arbitration primitives to reach the target memory bank. Similarly, memory responses propagate through arbitration and routing levels to reach back processors. In the Mega-Leon architecture, processors can synchronize by means of standard read/write operations at some memory-mapped registers providing test-and-set semantics (hardware semaphores). Moreover, the cores are connected to an AHB shared bus system to access the external modules (external memory, ROM, debug support unit and etc.).

The Mega-leon architecture enables program acceleration using dedicated hardware processing units (HWPUs). Similar to processors, in our proposal these HWPUs are also connected to the TCDM through the same network. Processors can program and configure HWPUs through a dedicated MoT interconnection. We will describe the details of HWPU architecture and the related design flow in the next sections.

B. HWPU design

In our proposal for HWPU design and integration, processors and accelerators directly communicate through a shared memory. Thanks to the logarithmic interconnection network and the multi-banked, multi-ported TCDM design, this communication takes place through a fast L1 memory module, and it avoids the costly data movements typically required when processors and accelerators have distinct memory spaces (e.g., like GPUs). Moreover, programmers are not required to entirely rewrite the accelerated software to explicitly deal with these transfers, as in our approach is only necessary to substitute the accelerated parts of code with simple API functions for offloading (see Section III-C).

Many HLS tools are currently available to easily create HDL code for our HWPUs. However, to efficiently design the shared-memory accelerators that we propose it is necessary to interface these HDL modules to our logarithmic network. To achieve this goal in an effective manner, and to allow to easily instantiate different HWPUs, we designed a wrapper to provide proper handshaking between the HDL module and the network. As a HLS tool we use Mentor's CatapultC [17]. We will refer to HDL modules generated by CatapultC as *CatC* modules in the following. Our generic wrapper design consists of three main blocks: the *CatC* module, the *Slave* module, and the *Synchronization* module. Its simplified block diagram is shown in Figure 3. The *Slave* module includes a register file and a controller for reading or writing the register file. The register file contains eighteen 32-bit registers used to initiate an offload sequence from a processor. More specifically, when a processor wants to start a HWPU, it is responsible for writing the base address of input/output data to the HWPU. These data items reside in the TCDM, so the processor only needs to communicate at which address they can be found. Currently, the register file can host up to 8 input data, plus 8 output data (16 registers). Once this is done, the processor can start actual HWPU execution by writing to a *trigger* register. Synchronization is achieved by inspecting the content of a *working* register. The latter shows the status of the HWPU (idle/busy) and its value is one when the HWPU is working. Figures 4 and 5 show timing diagrams for triggering HWPU execution and synchronizing with it from a processor. It can be seen that both operations complete in just one clock cycle.

The *Synchronization* module is responsible for interfacing the port(s) of the *CatC* module to master port(s) matching the interconnection network protocol. Handshaking and address translation is also implemented in the *Synchronization* module. Specifically, an address translation block provides the correct

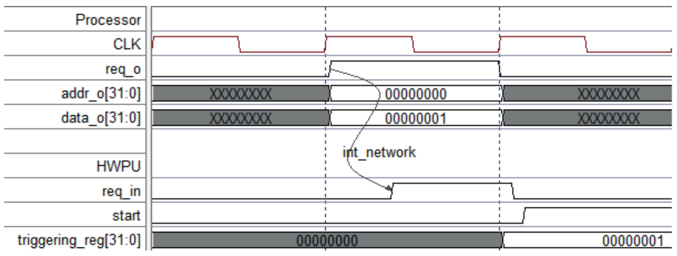


Fig. 4: Timing diagram for triggering HWPU execution

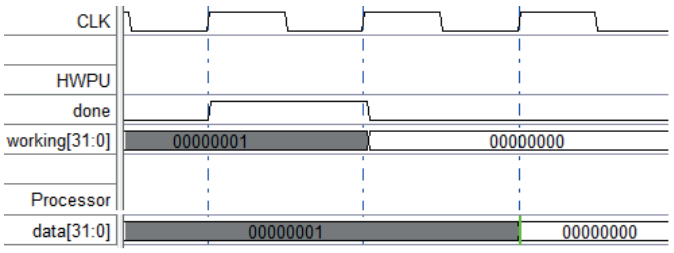


Fig. 5: Timing diagram for synchronizing with the HWPU

physical address to read/write target data through the master port of the HWPU. Indeed, the *CatC* module relies on a different memory map for its inputs and outputs, which needs to be remapped onto the global address space seen by processors. For this purpose, the base address of read/write data is fetched from the register file, to which a proper offset is added based on data type size generated during the synthesis of the *CatC* module (its internal memory map). Once the final address for the read/write operation has been computed, the corresponding transaction forwarded to the master port must match the credit-based protocol used by the network. The master which wants to initialize a transfer must assert a request signal (*data_req*). If the *data_stall* signal is raised in response, the transaction can not be handled. To accomplish the operation it is necessary to keep the request and data signals on the port. Write requests can be successfully completed once the *data_stall* signal becomes low. Read requests are split into request and response phases. The *data_stall* and *data_r_valid* signals are used by LOG INTC protocol to control these two phases. During the request phase our *Synchronization* module checks the *data_stall* signal before raising the request. If it is high, the module should be stopped. For this purpose, we bind the *∼data_stall* signal to the *enable* signal of the *CatC* module. However, when *data_stall* is high the response phase could still take place in parallel if *data_r_valid* is also high. We thus use a register (REG) to store the incoming valid data and a MUX to drive the correct input depending on the values of *data_stall* and *data_r_valid*. After raising the request, the CatC module samples a valid response data when the *wait_n* input is high. *wait_n* is thus bound to *∼data_stall*.

C. Software Infrastructure

To simplify the process of offloading jobs to the HWPU from the software we developed two sets of simple APIs to control HWPUs and synchronize core and HWPU execution.

TABLE I: List of Synchronization APIs

Function prototype	***Brief Description***
void BARINIT (int BARRIER_ID)	Initialize barriers
void BARRIER (int BARRIER_ID, int NUM_PROC)	Barrier synchronization
void WAIT (int LOCK_ID)	Acquire a lock
void SIGNAL (int LOCK_ID)	Release a lock

TABLE II: List of APIs for HWPU control

Function prototype	***Brief Description***
void WriteToHWPU (int HWPU_ID, int ADDR, int DATA)	Writes DATA in the register file of HWPU_ID at the specified address ADDR
void TriggerHWPU (int HWPU_ID)	writes *one* in TRIGGER register to start the HWPU
int IsWorkingHWPU (int HWPU_ID)	Reads the WORKING register to check if the HWPU is executing
int offload_HWPU (int HWPU_ID, int addr1, .. , int addrN)	a complete offload sequence to the HWPU

The first API, shown in Table I, provides primitives for *locks* and other inter-core synchronization mechanisms (e.g., barrier synchronization). These functions are implemented on top of the *test-and-set* memory in our architecture.

The second set of APIs, shown in Table II, provides routines to write parameters into the register file of the HWPU (`WriteToHWPU`), start execution (`TriggerHWPU`) and check termination (`IsWorkingHWPU`). We also provide a `offload_HWPU` function, which simplified code is shown in Fig. 6.

```
#define REG_ADDR1 0x4
...
#define REG_ADDRn 0x15
#define LOCK 0x0

int offload_HWPU (int ID, int ADDR1,...,int ADDRn)
{
  /* Acquire a lock*/
  WAIT (LOCK);

  if (!IsWorkingHWPU(ID))
  {
    /* base address for read operations*/
    WriteToHWPU(ID, REG_ADDR1, ADDR1);
     ...
    WriteToHWPU(ID, REG_ADDR7, ADDR2);

    /* base address for write operations*/
    WriteToHWPU(ID, REG_ADDR8, ADDRn-1);
     ...
    WriteToHWPU(ID, REG_ADDR15, ADDRn);

    /* triggering HWPU*/
    TriggerHWPU(ID);

    SIGNAL (LOCK);
    return 1; /* successful offload */
  }
  else
  {
    SIGNAL (LOCK);
    return 0;  /* unsuccessful offload */
  }
}
```

Fig. 6: Offloading a job

This API abstracts the details of completing a programming sequence, and goes through the following steps:

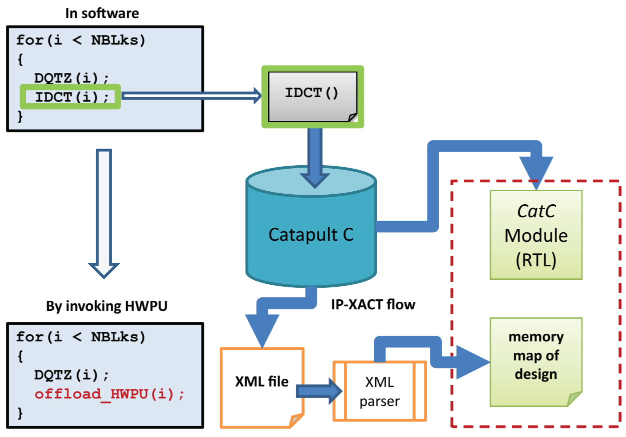
- The processor acquires a lock to ensure exclusive ownership of the target HWPU
- The processor checks the status of the HWPU through the `IsWorkingHPU` API.

- When the HWPU it free, the processor writes the base addresses of I/O data by invoking `WriteToHWPU`.
- The HWPU is started through the `TriggerHWPU` API.

While executing, the HWPU operates on I/O data directly in the TCDM. When the HWPU finishes its job, it sets the `working` register to zero. The processor synchronizes with the HWPU by calling the `IsWorkingHWPU` function.

D. Tool Flow

Identifying the most performance- and energy-efficient implementation of an accelerator-based system requires design space exploration. For this purpose, High level synthesis (HLS) tools are widely used to automatically generate HDL code from C programs. However, typically this requires significant effort for HWPU development and integration, as well as application re-writing to deal with processor-to-HWPU communication and synchronization. We have described in the previous section the software infrastructure that we developed to simplify HWPU programming. It is also important to underline that thanks to the shared-memory accelerator model that we propose, the application code needs not be heavily modified, since all the communication (and thus data management) is implicit in the model. In our framework, candidate regions of code for acceleration are extracted from a C program and fed to CatapultC. These regions of code are simply replaced by calls to the offloading API. Besides generating the RTL model of the target HWPU logic, Catapult C also generates as an output a *xml* file containing an internal memory map for I/O data. We parse this *xml* file to automatically generate HDL parameters (VHDL generics) for the address translation block in the *Synchronization* module of our wrapper. The tool flow for using Catapult C is shown in Fig.7.



```
for(i < NBLks)
{
   DQTZ(i);
   IDCT(i);
}
```

```
for(i < NBLks)
{
   DQTZ(i);
   offload_HWPU(i);
}
```

Fig. 7: Tool flow for HWPU instantiation

IV. Experimental Results

In this section we evaluate the proposed architecture. We consider two target applications for acceleration, namely a JPEG decoder and a Scale Invariant Feature Transform (SIFT), extracted from the OpenCV library [22]. First, we identified the main computational kernels of the two applications, and profiled their execution time to identify the best candidates for HW acceleration. JPEG consists of four kernels, *huffman DC*, *huffman AC*, *dequantization* (*DQTZ*) and *inverse DCT* (*IDCT*). *IDCT* and *huffman AC* take 45.40% and 42.49% of the overall execution cycles, respectively. However, *IDCT* is better suited for HW implementation, due to the data-parallel nature of its computational pattern. SIFT includes four main kernerls: image *upsampling* and *downsampling* (*SMPL*), *gaussian filtering* (*GAUS*), *difference of gaussians* (DOG) and *feature extraction*. More than 80% of the overall execution time is spent in the *GAUS* kernel, which we selected for HW acceleration. In fact, we consider three different instances of the *GAUS* HWPU, which use different sizes of the neighbouring pixel mask for the computation (7×7, 11×11, 13×13).

In the following discussion we first provide synthesis results (area, power) for the target platform in Section IV-A. Then, in Section IV-B we discuss different schemes for parallelization and acceleration of the applications. For these experiments we consider two main performance metrics, namely execution cycles and performance/area/watt.

A. Synthesis Results

To evaluate the performance and hardware cost of our approach we use a commercial RTL simulator. Synopsys design compiler [19] is used for hardware synthesis and designs are mapped on CMOS 65nm technology from ST-Microelectronics. We consider two platforms, the reference *Megaleon* architecture described in Section III-A, plus an enhanced platform with one IDCT HWPU and three GAUS HWPUs (7×7, 11×11, 13×13). Table III present the synthesis results for these architectures, in terms of area and power. It is possible to see that area and power overheads for our HWPUs are always within 6% of the total. Also, the overal area for the wrapper is 4335 cells in ST65nm.

TABLE III: Synthesis and power results

	Megaleon				
	cores	***LOG INTC***	***Mems***	***Others***	
Area(mm^2)	1.90	0.16	1.76	0.29	
Power(mW)	92.54	2.30	2.39	11.08	
	HWPU				
	IDCT 1	***GAUS***	***LOG1***	***LOG Periph***	***OVH(%)***
Area(mm^2)	0.098	0.07	0.01	0.01	**4.55**
Power(mW)	2.09	3.72	0.31	0.59	**5.84**

B. Run time evaluations

In this section we present performance results for different partitioning schemes of applications on processors and HWPUs. Our experiments are aimed at assessing system scalability for two different metrics: execution time and performance/area/watt (p/a/w). To estimate p/a/w scaling we consider a fixed area (the overall platform area shown in Table III), while the power scales with the core count (we consider power gating for the unused processors). We focus on the data-parallel kernels from our applications, and consider two different partitioning approaches to evaluate the impact of HW acceleration as compared to pure parallelization:

(i) ***SW:*** All kernels are parallelized in SW.

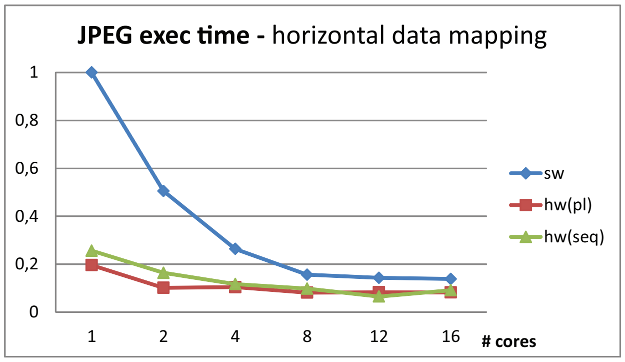


Fig. 8: Execution time scaling for DQTZ+IDCT. Horizontal data mapping.

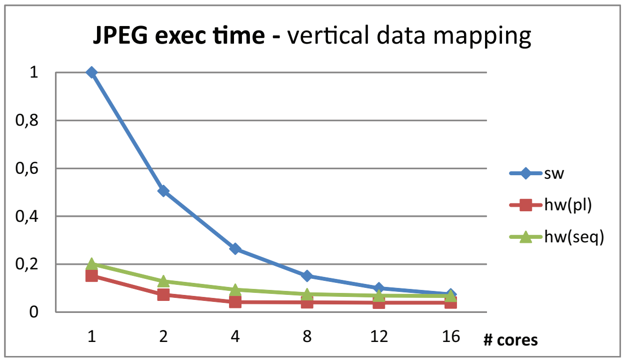


Fig. 9: Execution time scaling for DQTZ+IDCT. Vertical data mapping.

(ii) ***HW:*** Candidate kernels for HW acceleration (see previous section) are offloaded to a HWPU. Processors offload jobs to the HWPU in parallel, and wait when it is busy. The remaining kernels are parallelized in SW.

1) JPEG: Regarding *JPEG*, both *DQTZ* and *IDCT* can be data-parallelized at the macro-block (MB) level. The computation takes place in two consecutive loops, where processors can concurrently access different blocks (one block per loop iteration), as no data dependencies are present. The simplest approach for SW parallelization is to distribute loop iteration among processors. First the whole DQTZ parallel loop is executed, then the IDCT parallel loop. We will in the following refer to this configuration as *sequential* (*seq* in the plots). Here, during the *IDCT* loop several processors try to offload their assigned computation to a single HWPU. Thus the execution of IDCT blocks is sequentialized on the HWPU, no matter how many processors are assigned to work in parallel in the loop. To avoid the sequentialization of HWPU usage implied in this parallelization scheme, we consider a second approach where within a single loop we pipeline the execution of DQTZ and IDCT over each macro-block. In this way, after offloading its job to a HWPU a processor does not stay idle waiting to offload the successive IDCT block, but it rather actively works on the next DQTZ block. This parallelization scheme is explained in Figure 10. We will refer to it as *pipelined* (*pl* in the plots) in what follows.

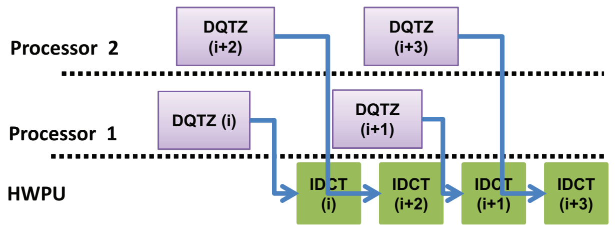


Fig. 10: Pipeline execution of DQTZ and IDCT

Another exploration that we make with this experiment regards data placement. Each loop iteration independently accesses distinct macroblocks (MB), and each MB consists of 64 pixels, which are by default mapped horizontally over 2 TCDM rows (the TCDM has 32 banks, uses word interleaving, and each pixel is represented with a word). Thus, each MB occupies 2 full TCDM rows, and consecutive blocks are stacked with this same alignment. Consequently, if different processors/HWPU access to distinct blocks in parallel, they will be in practice trying to concurrently access pixels that are mapped onto the same bank. This situation is likely to lead to banking conflicts. We have thus considered also a second data mapping, where each MB is vertically laid within a bank. Groups of 32 consecutive MBs are thus entirely placed onto distinct banks, which removes the source of contention.

Figures 8 and 9 show the execution time scaling for this experiment under horizontal and vertical data placement, respectively. The plots compare ***SW*** and ***HW*** approaches, with partitioning schemes *seq* and *pl*. Focusing on the ***SW*** approach, it is possible to confirm the impact of vertical data placement on conflict reduction. For horizontal placement the parallelization scheme scales only up to 8 cores, whereas for vertical placement scaling continues up to 16 cores. Regarding the ***HW*** approach, when the number of cores in the system increases we see diminishing returns, since during the *IDCT* loop several processors are trying to offload their assigned computation to a single HWPU. Thus the execution of IDCT blocks is completely sequentialized, no matter how many processors are involved in the loop.

In Figures 11 and 12 we show performance/area/watt (p/a/w) for this same experiment, considering horizontal and vertical data mapping, respectively. In addition to *pl* and *seq* ***HW*** configurations we also consider here *ideal* values for these acceleration schemes (*ID(pl)* and *ID(seq)* in the plots). The bars for *ideal* acceleration schemes are computed by considering HWPU execution time when no data conflicts are present. Sequentialization of HWPU offloading from multiple processors is considered when modeling ideal values. P/a/w values on the Y axis for the two plots are normalized to the highest (i.e., *ID(pl)* for 4 cores, vertical data mapping). A first consideration is that data conflicts on the horizontal data layout have a big impact on p/a/w. First, the highest ideal values that can be obtained when this layout is considered are smaller. Second, there is a big gap between the ideal values and the actual ones. It is possible to see that vertical

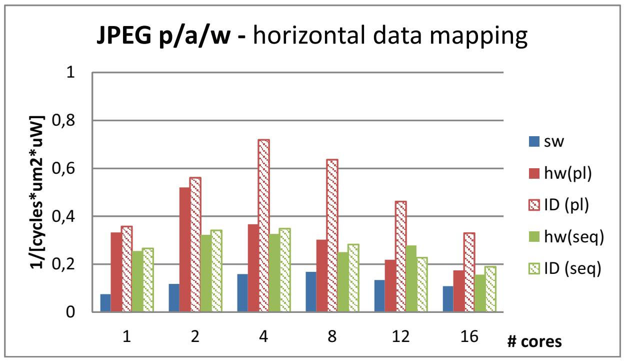


Fig. 11: Performance/area/watt for DQTZ+IDCT. Horizontal data mapping.

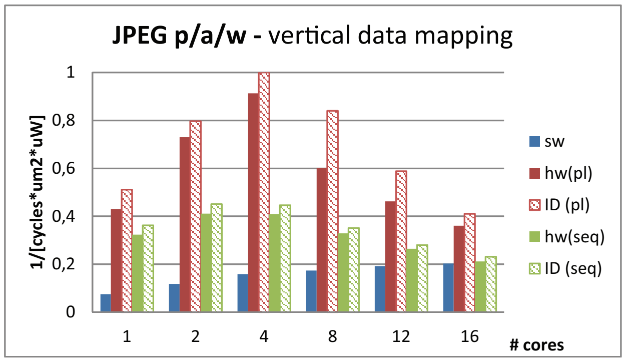


Fig. 12: Performance/area/watt for DQTZ+IDCT. Vertical data mapping.

mapping brings actual values much closer to the ideal. The best results when horizontal layout is considered can be obtained with 2 processors, with an overall speedup vs ***SW*** of $4.42\times$. When vertical layout is used, the best p/a/w is achieved with 4 processors, with an overall speedup vs ***SW*** greater than $6\times$.

2) SIFT: Regarding SIFT, our approach to application partitioning is the following. The algorithm goes through the three stages in sequence: SMPL, GAUS, DOG. SMPL is executed three times on the input images and produces 3 scaled output images. Three GAUS kernels (with mask sizes of 7, 11 and 13, as explained before) produce 3 transformed images. DOG finally executes twice to produce the pixel-by-pixel difference of these three images. All the three kernels take place within loops that can be parallelized by distributing iterations among available cores. At the end of each loop a barrier is required to preserve dependencies. For the ***SW*** approach we consider multi-level parallelization. Specifically, we assign each repetition of a kernel to a thread (e.g., GAUS7×7, GAUS11×11 and GAUS13×13 are assigned to three different threads). Then, the loops inside each repetition are parallelized among additional available threads. For this reason, we consider scaling the processor count by multiples of three. When there are $3\times R$ processors available, each of the three loops (GAUS7×7, GAUS11×11 and GAUS13×13) will be parallelized over R cores. We call this parallelization scheme *symmetric* throughout the rest of this section. When using the ***HW*** approach, during each *GAUS* repetition all the R processors are trying to offload their iteration to the associated HWPU, and they will wait in case it is busy. We thus consider an additional ***HW-SW*** approach, where if the HWPU is busy the calling processors execute the job in software.

Figure 13 shows execution time scaling for this experiment. First, it should be noted that with single core configuration (i.e., absence of data conflicts), the ***HW*** solution can only speed-up execution time by $2\times$ w.r.t. ***SW***. This is due to the fact that the GAUS kernel is heavily memory-bound, thus most of its execution time cannot be accelerated. The ***HW*** approach only scales from 1 to 3 cores, because in the first case the three HWPUs are invoked in sequence, whereas in the second jobs are offloaded in parallel. However, beyond 3 cores the ***HW*** approach does not show improvements, because of the sequentialization of offload requests. The break-even point is arond 6, 7 cores, where the ***SW*** approach starts doing better than the ***HW***. Overall, the ***HW-SW*** approach enables further improvements. Looking at Figure 14, the performance/area/watt results confirm that the best design point for this experiment is the ***HW*** configuration with 3 cores.

We do not repeat the experiment with vertical data placement for *SIFT*, as the access pattern in this program is such that all the neighbouring elements to a pixel are accessed, which results in conflicting accesses even if data is placed vertically. Instead, we consider another experiment where the number of processors assigned to kernels *GAUS7×7*, *GAUS11×11* and *GAUS13×13* is *asymmetric*. Indeed the execution time for the three kernels is different, with *GAUS13×13* being 3 times slower than *GAUS7×7*. Of course we do not expect to see any benefits in this *asymmetric* parallelization for the ***HW*** approach, as in any case all offload requests will be sequentialized in both cases. However, we expect *asymmetric* core distribution to have a beneficial effect on the ***HW-SW*** approach.

Execution time and performance/area/watt results for this experiment are shown in Figures 15 and 16, respectively. The numbers above the **#cores** on the X-axis describe the processor count assigned to each of the three GAUS kernels. As expected, both the ***SW*** and ***HW-SW*** approaches improve, while the ***HW*** solution is unchanged. Eventually, identical peak performance/watt is achieved for ***HW*** with 3 processors, or for ***HW-SW*** with 6 processors.

V. Conclusion

In this paper we presented an architecture which integrates hardware processing units (HWPU) in a tightly-coupled shared memory multi-core cluster. Thanks to the shared memory processor-to-HWPU communication model we avoid the cost for frequent data transfers typically implied by accelerator-based programming models. Moreover, the shared memory abstraction also makes it easier to write accelerated programs, since data needs not be explicitly managed among separate memory spaces. We designed a generic communication interface for HWPUs in the form of a wrapper where the core functionalities can be easily synthesized through a HLS tool.

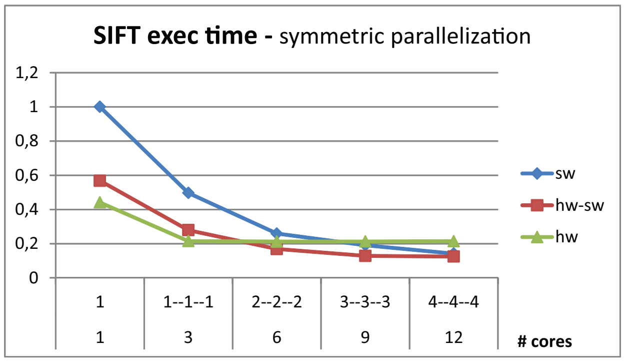


Fig. 13: Execution time scaling for SIFT. Symmetric parallelization.

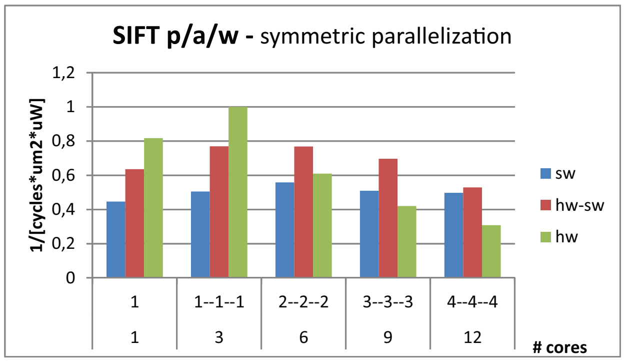


Fig. 14: Performance/area/watt for SIFT. Symmetric parallelization.

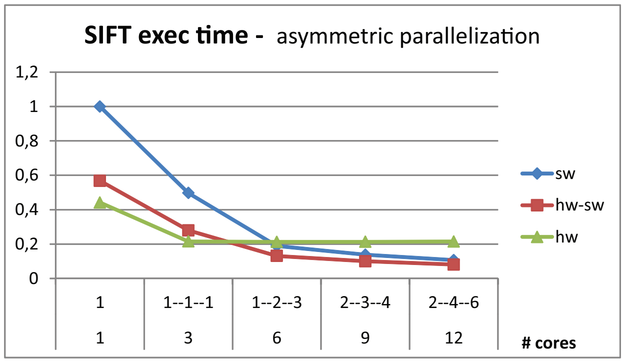


Fig. 15: Execution time scaling for SIFT. Asymmetric parallelization.

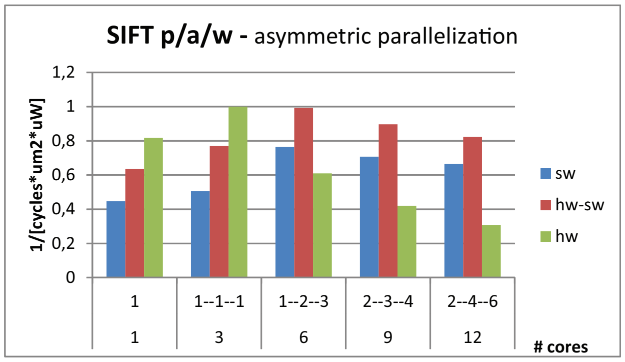


Fig. 16: Performance/area/watt for SIFT. Asymmetric parallelization.

In addition, we provide a SW API to facilitate the development of accelerated programs. Our experimental results demonstrate the effectiveness of our proposal.

Acknowledgment

This work was supported by project FP7 VIRTICAL (288574), funded by the European Community.

References

[1] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. Proceedings of the IEEE, 96(5), May 2008.
[2] R. Dolbeau, S. Bihan, and F. Bodin. HMPP: A hybrid multi-core parallel programming environment. Technical report, CAPS enterprise, 2007.
[3] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: exploiting graphics processing units to accelerate distributed storage systems," in Proceedings of the 17th International Symposium on High Performance Distributed Computing, Boston, MA, June 2008.
[4] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in Architectural Support for Programming Languages and Operating Systems, Pittsburgh, PA, March 2010.
[5] M. Becchi, S. Byna, S. Cadambi, and S. Chakradhar, "Data-aware scheduling of legacy kernels on heterogeneous platforms with distributed memory," in SPAA: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures, June 2010.
[6] ST Microelectronics and CEA. Platform 2012: A Many-core programmable accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology. 2010
[7] A. Rahimi, I. Loi, M.R. Kakoee, L. Benini "A Fully-Synthesizable Single-Cycle Interconnection Network for Shared-L1 Processor Clusters ," in Proc. of the ACM/IEEE DATE, 2011.
[8] Gelado I., Kelm J.H., Ryoo S., Navarro N., Lumetta S.S., Hwu W.W. CUBA: An Architecture for Efficient CPU/Co-processor Data Communication. ICS, June 2008.
[9] Wang P., Collins J.D., Chinya G. N., Jiang H., Tian X., Girkar M., Yang N. Y., Lueh G., Wang H. Exochi: Architecture and programming environment for a heterogeneous multi-core multithreaded system. PLDI 2007.
[10] Plurality, Ltd. The hyperCore architecture, white paper, January 2010.
[11] N. Bayer, A. Peleg, "Shared memory system for a tightly-coupled multiprocessor," Pub. no. WO/2009/060459, 2009.
[12] Kronos Group. The OpenCL 1.1 Specifications. www.khronos.org/registry/cl/specs/opencl-1.1.pdf , 2010.
[13] The Portland Group. PGI Acceleration Programming Model for Fortran and C. www.pgroup.com/ lit/whitepapers/pgi accel prog model 1.3.pdf , 2010.
[14] NVIDIA. Next Generation CUDA Compute Architecture: Fermi - WhitePaper. www.nvidia.com/content/PDF/ fermi white papers/NVIDIA Fermi Compute Architecture Whitepaper.pdf , 2010.
[15] J. Nickolls and W. J. Dally, "The GPU computing era," IEEE Micro, vol. 30, no. 2, pp 56 - 69, April 2010.
[16] http://www.gaisler.com
[17] T. Bollaert, Catapult synthesis: A practical introduction to interactive C synthesis, in High-Level Synthesis: From Algorithm to Digital Circuit, P. Coussy and A. Morawiec, Eds. Heidelberg, Germany: Springer, 2008.
[18] AMD Inc. Fusion series. http://www.amd.com/us/products/technologies/ fusion/Pages/fusion.aspx.
[19] www.synopsys.com/tools/
[20] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, D. Burger, "Dark silicon and the end of multicore scaling", Proceeding of the 38th annual international symposium on Computer architecture, June 04-08, 2011, San Jose, California, USA
[21] N. Goulding-Hotta, J. Sampson, G. Venkatesh, S. Garcia, J. Auricchio, P. Huang, M. Arora, S. Nath, V. Bhatt, J. Babb, S. Swanson, and M. Bedford Taylor, "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future", Micro, IEEE 31(2):86 -95, march-april 2011.
[22] G.R. Bradski, A. Kaehler, "Learning OpenCV: Computer vision with the OpenCV library", OReilly Press, Cambridge, MA, 2008