Transparent Near-Memory Computing with a Reconfigurable Processor

Fabian Lesniak[0000−0001−8302−6086], Fabian Kreß[0000−0002−1700−5778], and Jürgen Becker[0000−0002−5082−5487]

Karlsruhe Institute of Technology

Abstract. Data intensive applications like machine learning or big data analysis have stressed the requirements on memory subsystems. They involve computational kernels whose performance is not limited by the algorithmic complexity, but by the large amount of data they need to process. To counteract the growing gap between computing power and memory bandwidth, near-memory processing techniques have been addressed to improve the performance in such applications significantly. In this paper, we leverage a general purpose processor extended with a reconfigurable fabric to execute hardware-accelerated instructions. This fabric features a high-bandwidth memory interface to the nearest memory controller, allowing for greatly increased bandwidth compared to the standard system bus. We introduce region-based data processing, which allows to trigger operations by merely storing data and is especially suitable for large many-core designs. We show two different approaches to trigger the architecture for near-memory operations, one using interrupts for software-assisted processing and one directly interfacing with the hardware accelerator. Our evaluations show a performance gain of 72% on SAD kernels, with memory performance improved by 48%. Benchmarking AES encryption, we can show a speed up of 70%.

1 Introduction

The demands on processor architectures are constantly increasing and one of the major trends of the industry to meet these increasing requirements is the further development and integration of application-specific circuits. Several accelerators for all types of applications are integrated on a single chip. Recent developments in data-centric applications (e.g. large data, AI, media) have shifted the focus to efficient data processing. Improvements in industry include the implementation of SIMD instructions into the instruction set, as the SSE and AVX extensions do, and developments in memory technology increasing data rates as demonstrated with High-Bandwidth Memory (HBM) and Hybrid Memory Cubes (HMCs). Even though these improvements increased processing power and bandwidth, many workloads still saturate available memory bandwidth. This especially becomes important in large-scale systems, where memory transfers span longer distances and multiple levels of hierarchy.

Moving processing elements closer to memory is one common approach to overcome these bandwidth limits. The design space is thereby largely increased, as

the designer has to choose multiple parameters when implementing an accelerator for near-memory computing (NMC):

- Determine which operations to accelerate based on workload analysis
- Positioning the accelerator within the performance, power and area triangle
- Deciding on the basic architecture, whether to use a reprogrammable processor, FPGA or ASIC implementation

Choosing the basic architecture comes down to a trade-off between flexibility and performance. While using a processor brings high flexibility, its performance is considerably lower than a hardware implementation. In opposite, a specialized ASIC implementation is inherently unalterable but provides highest performance. Reconfigurable hardware (FPGA) tries to combine the best of both worlds, typically reaching higher performance than CPU-based solutions and allowing reconfiguration. However, the performance of an ASIC can't be matched and FPGA designs are generally more difficult to build than a software solution. The approach presented in this paper features a combination of a CPU with a reconfigurable FPGA partition, thus allowing to execute control-flow heavy software as well as high performance RTL accelerators close to memory.

The remainder of the paper starts with a presentation of related work (Sect. 2). It is followed by a presentation of our concept of region-based NMC, including both the software-assisted and the direct accelerator approach (Sect. 3). Based on that, we present our reference implementation (Sect. 4) and results from different benchmarks are shown (Sect. 5). Finally, a conclusion is drawn and outlook on future work is made (Sect. 6).

2 Related work

NMC is a common approach to improve performance of data-centric workloads. As many different approaches have already been published, this section classifies some of these approaches and compares them to our design.

An important property of NMC implementations is the type of memory the system is working on. Depending on the workload it can be feasible to place accelerators close to bulk storage, as shown with *Summarizer* [8]. The authors provide interfaces to offload work from the host processor to a SSD controller. In contrast, in-memory approaches like *iPIM* [5], which features an in-memory image processing accelerator, profit from very high memory bandwidth while waiving flexibility.

Apart from the memory target, the fundamental design of the accelerator itself can be distinguished. Several approaches, as previously mentioned *Summarizer*, use CPUs which are placed close to memory to perform operations. The benefits are high flexibility as software can easily be exchanged and low hardware costs due to the broad availability of processors. However, some approaches chose FPGA or ASIC based accelerators for even higher processing power. Corda et. al. [2] demonstrated large performance benefits when using FPGAs with direct

memory access for fast fourier transform, outperforming both CPU and GPU based approaches.

Kang et. al. present an approach to leverage existing data wires of a STT-MRAM chip to perform bitwise logic operations during readout [7]. While accelerating single instructions by executing it during memory access, this type of accelerator is an enhancement to an existing processor and in principle unable to perform large operations on its own.

Reconfigurable processors [3] try to bridge the gap between the flexibility of processor-based approaches and the processing power of specialized hardware. They allow to use special instructions (SIs) to make use of the reconfigurable fabric, resulting in faster execution compared to a pure software-based implementation. Previous work also shows how to detect code suitable for SI-based acceleration and generate RTL code for appropriate accelerators automatically [6], reducing hardware design efforts.

In our design, we use such an reconfigurable processor and investigate the advantages of placing it close to memory. This allows to have a dedicated, high-bandwidth memory interface, but retain the performance and flexibility of a monolithic processor.

3 Concept

This section will first introduce the architecture of our near-memory accelerator (NMA) containing a reconfigurable processor. Afterwards, our region-based approach for transparent operation on the memory bus is discussed and its advantages are highlighted.

3.1 Accelerator architecture

The core component is a reconfigurable processor which is placed close to memory. Figure 1 shows a high-level view of an example system containing multiple general purpose CPUs, peripherals and memory. The memory has a high-bandwidth connection to the NMA, which consists of a processor core and a reconfigurable fabric. Processor and FPGA fabric are connected to allow special instructions to be executed in the reconfigurable partition.

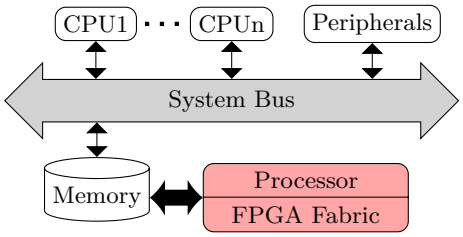

Fig. 1: NMA consisting of processor and FPGA fabric in a multi-processor system

By containing a standard CPU core, it can directly execute code without requiring RTL design. The key advantage is the low barrier for porting an existing algorithm to the accelerator: software can easily be adapted to run on the NMA. Even without using the reconfigurable portion of the accelerator, the software can benefit from improved memory performance. The generic concept does not require a specific processor architecture, thus allowing the designer to choose one matching the design constraints. The processor however needs to provide an interface for the reconfigurable fabric, allowing custom instructions to call the hardware accelerator.

The reconfigurable fabric is an FPGA-based hardware accelerator. It allows to take over computationally intensive tasks from the near-memory processor, thus further boosting performance of the NMA. Implementing an entire algorithm in HDL is often a very time-consuming task. Our approach allows to make use of the interaction between processor and hardware accelerator, as merely the expensive parts of an algorithm have to be implemented in hardware to achieve significant performance gains. Other work, such as initialization and control flow tasks, can still be fulfilled in software. Compared to memory-mapped accelerator interfaces, the close coupling between processor and reconfigurable fabric allows to easily engage the accelerator by using an appropriate special instruction.

Reconfiguration features enable to use the accelerator fabric for varying tasks. The required accelerator configuration can be loaded when required, falling back to a software implementation if the accelerator is shared and other tasks have higher priority. Additionally, the NMA can adapt to unprecedented use cases in the future, still allowing acceleration of updated software where non-reconfigurable approaches need to fall back to software-only processing.

3.2 Region-based near-memory computing

The reconfigurable fabric is primarily controlled by software, as it requires a special instruction to operate. All clients using the NMA are thus required to synchronize with the software running on the accelerator. Typical schemes for sending a command to the accelerator are remote procedure calls (RPCs), requiring a communication channel to be set up before. To simplify and reduce communication effort between client and accelerator, we propose to make the accelerator transparent to memory access: The memory, which the accelerator operates on, is fully accessible via the bus and thus accessible to all clients. A client can define regions within that memory, consisting of a starting address, size of the region and an operation. By watching the memory bus, the accelerator can identify access to such a region and trigger the associated operation. This trigger can decide whether enough data is available, depending on the selected operation, and start executing that operation automatically. Thus, the effort to use the NMA is reduced to configuring an appropriate region once. Afterwards, it is sufficient to stream input data to the respective address in memory and read output data after processing is done.

As in our concept the NMA contains both a CPU core and an accelerator fabric, this yields two possibilities for the NMA to be triggered:

Interrupt-based trigger The processor core receives an interrupt after a region received enough data. The NMA starts processing in software and can use the hardware accelerator if suitable. We call this the *software-assisted mode*, as algorithms can be partly accelerated by hardware and run the remaining operations in software. The software-assisted mode can make use of reconfiguration by loading different accelerated instructions during one operation.

Direct hardware trigger The hardware accelerator is directly started without software intervention. As no software overhead occurs, this approach allows very low latencies. The NMA processor can even execute other code in parallel, as the hardware accelerator does not interfere with the processor resources. However, the operation associated with the region needs to be fully implemented in hardware. This may not be possible for complex algorithms due to limited hardware resources and the lack of reconfiguration.

The region-based NMC approach can be extended to allow overlapping regions. Thus, multiple algorithms can be applied consecutively to the same data without requiring to copy data in between. An example of overlapping regions is shown in Fig. 2, where two regions are filled with data to be sorted. First, the quicksort algorithm is applied to each region separately. Second, the mergesort algorithm can be used to merge both regions and obtain a fully sorted array.

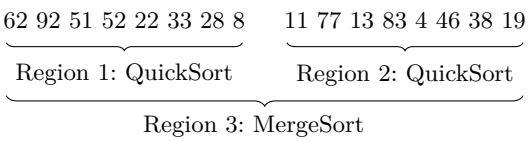

Fig. 2: Combining sorting algorithms using overlapping regions

3.3 Software interface

The region-based concept simplifies the steps necessary in software to use the NMA. Traditional approaches use RPCs or memory-mapped control registers to configure the accelerator and fine-grained execution control is handled by a driver in software. In contrast, our approach requires configuration once, afterwards data transfer to the configured region is sufficient. Figure 3 shows the interaction between client, NMA and memory.

The steps required to use the accelerator can be described as follows:

1. The client registers a region for use, including size and an associated operation. Region conflicts with other users need to be avoided, however, the memory management of the operating system can be used to guarantee unique buffer allocation.

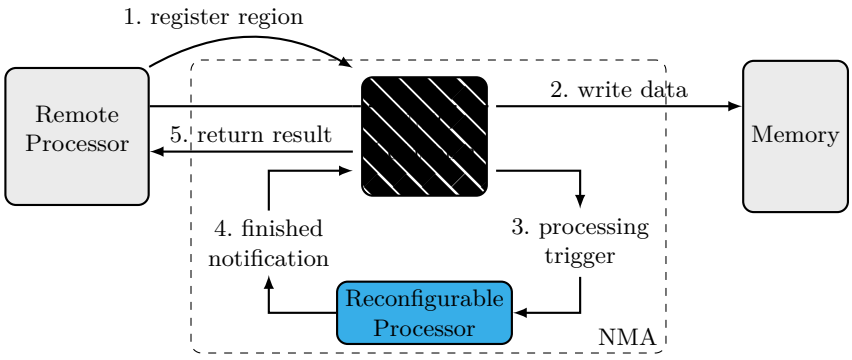

Fig. 3: Configuring regions and using the NMA

2. Payload data is now written directly to memory, a call to `memcpy` or using DMA is sufficient.
3. The NMA observes memory writes and triggers the reconfigurable processor as soon as its processing condition is met. Either the reconfigurable fabric is started directly or the software-assisted mode is used, depending on the selected operation.
4. The reconfigurable processor notifies the controller when it has finished processing the region.
5. A result is returned to the client, if necessary.

Interaction with software on the client takes place in steps 1, 2 and 5 only. Steps 2 and 5 can be repeated arbitrarily to process further data. The client thus does not need to send any control commands after setting up the region, reducing communication with the NMA to a minimum.

3.4 Synchronization mechanisms

As clients do not control the accelerator directly, it is required to establish means for synchronization to decide whether a region is in use and when an operation is finished. Additionally, the memory regions need to be protected from modification while the accelerator is working on them. These requirements can be realized by extending the behavior on the bus system. As a bus slave, the accelerator can delay or deny requests if they are currently not appropriate.

The behavior of the accelerator differs for the following three states of a region:

Region idle Writing data is allowed, waiting for enough data to start processing. Reading can be denied if necessary, i.e. secret data waiting for encryption.

Region busy Both read and write requests stall until processing is done or a bus error is returned if non-blocking operations are preferred.

Region done Reading is allowed. Returning to idle automatically if all output data is read or manually by register write.

To allow continuous writing and reading of data, multiple regions can be allocated and used in a round-robin fashion.

3.5 Large-scale platforms

In recent years there has been a trend towards ever larger multi-core systems with heterogeneous memory architectures. To improve manageability of such platforms, many-core systems are usually divided into tiles. Therefore the notion of local and remote resources can be established, depending on where a program is currently executed. While accessing a local accelerator resource on the same tile is simple, accessing remote accelerators requires support by hardware and software. Software support in the operating system or runtime libraries uses on-chip communication mechanisms to configure the remote accelerator and transmit payload data.

In such a scenario, our transparent approach brings several benefits: While operating system features for inter-process communication can still be used for allocation and synchronization of accelerator access, moving data to and from the accelerator does not require intervention by the runtime system. Especially in partitioned global address space (PGAS) systems, data transfer solely involves the remote user and its DMA controller. This saves additional API calls for submitting a buffer to be processed, which can be costly depending on the communication mechanism used.

However, there is a tradeoff between copying data to an accelerated processor and processing without acceleration. Depending on the amount of data to process and the performance gain by hardware acceleration, it can be faster to process data without acceleration than moving it to the NMA memory prior to processing. Avoiding to store data locally and using memory backed by the NMA providently can reduce this effect.

4 Implementation

After introducing the key aspects of our concept, this section provides the hardware mechanisms that were implemented to evaluate the proposed architecture.

4.1 Overview

The *i*-Core Platform [3] is used as framework of the implementation as it already offers some features that match our concept. It provides a reconfigurable processor consisting of a LEON3 core and a reconfigurable fabric which is connected to the internal Scratchpad Memory (SPM). Since this fabric is attached to the execution stage of the processor, computations can be initiated using custom instructions. Moreover, the integrated bitstream loader allows for runtime reconfiguration of the fabric. This approach improves flexibility as various applications can run their specific accelerated instructions on the same reconfigurable fabric.

Nevertheless, the implementation of the proposed concept requires to extend the platform by modifying the processor itself and providing a new module called NMC Controller as shown in figure 4.

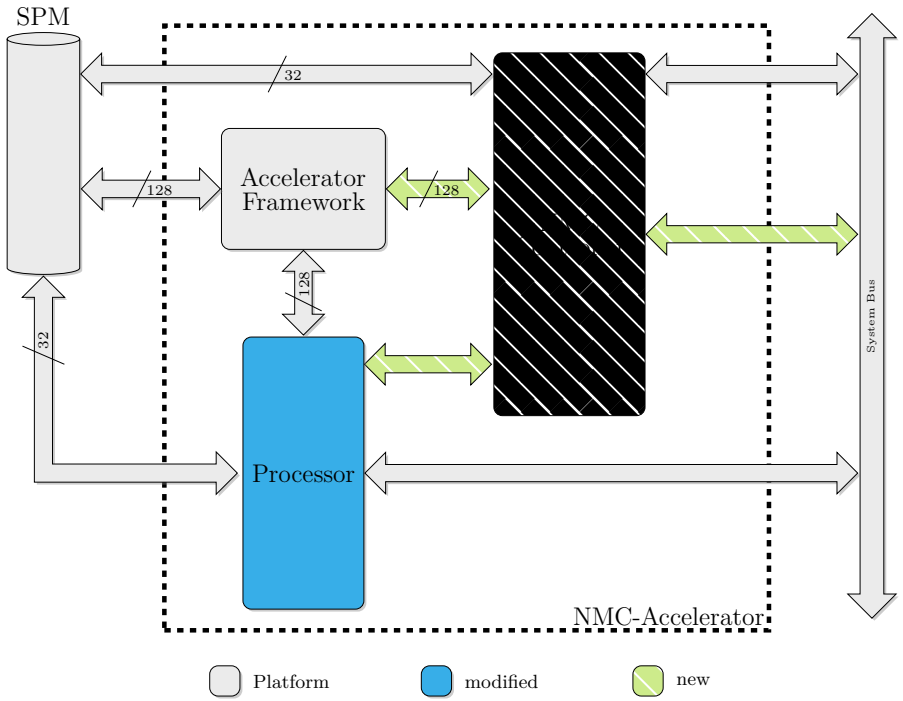

Fig. 4: Implementation Overview

4.2 Processor

Due to its ability to run a broad range of tasks the internal processor is a key element to implement a very flexible NMA architecture along with the reconfigurable fabric.

As a bus master, it can access not only the internal SPM but also other system components to load data if necessary. Therefore, clients do not have to provide all the data required to process the task if the missing values are accessible to the NMA internal processor.

To notify the processor about pending tasks in the software-assisted mode, it is directly connected to the NMC Controller. As our processor design allows customizable hardware traps, this feature is used to assert an interrupt each time the processor needs to start working on a region. Moreover, this approach allows the processor to execute unrelated tasks in parallel, which get interrupted automatically whenever the accelerator itself requires action. In our reference

implementation, the operating system treats the reconfigurable processor like a normal processor, as it is able to execute generic workload in parallel.

Beside a connection for interrupts, the processor also needs information about the selected operation and address range (region) to operate on. These values are directly wired from the NMC Controller into the execution stage of the processor.

Once the execution of the NMC task has finished, the processor transmits the result to the NMC Controller and returns to its previous state.

4.3 NMC Controller

The main component of our implementation is the NMC Controller. To realize the synchronization mechanisms (Sect. 3.4), it is placed between system bus and SPM to be able to monitor and control read and write access to the SPM. During execution, the NMC Controller can deny or delay write access to the SPM to prevent the NMC result from being affected by inconsistent values in memory. Moreover, in case of encryption algorithms it also allows to ensure that clients are not able to read the plaintext. When delaying a memory access, bus splits are used to avoid locking up the system bus. Figure 5 shows the architecture of the NMC Controller implementation.

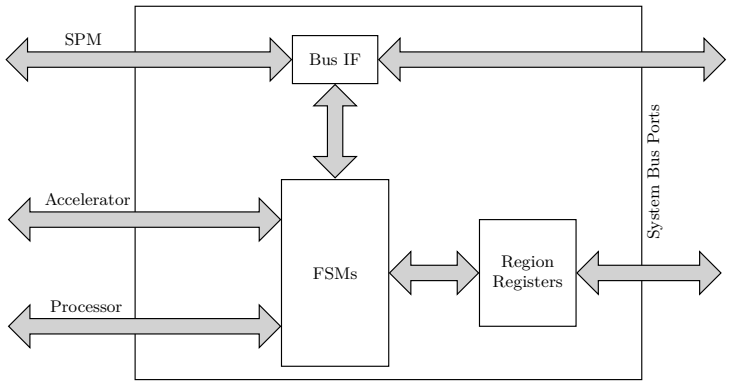

Fig. 5: Schematic of the NMC Controller

As soon as the address of a write access to the SPM matches the processing condition of an enabled region, the corresponding action is triggered. Thereby our implementation offers two different possibilities to execute the desired task. It can either raise an interrupt in case of the software-assisted mode or alternatively access the reconfigurable fabric directly using the COREFAB [4] interface. Originally designed to allow remote cores to execute SIs on the hardware accelerator, it allows to bypass the processor and directly execute a task in the accelerator framework. Therefore the NMC Controller uses this interface to run SIs on the reconfigurable fabric. Input parameters required by the accelerator framework such as operands and the SI to run have to be configured as part of the region

configuration prior to execution. Nevertheless, running SIs by the internal processor is still possible since the implementation of the extended framework comes with an arbiter to switch between local and NMC instructions. It also provides additional logic to merge instructions if both local and remote core request to run a SI at the same time but not on the whole fabric.

Since overlapping regions are allowed, a round-robin arbiter has to decide which NMC task is executed first in case of multiple regions raising a request at the same time. In addition, requests can be separated into one queue for each operating mode. Hence, in case of one region requesting a software-assisted task and another one asking for a task being directly executed in hardware, both requests can be initiated at the same time. We therefore implemented two independent finite-state machines (FSMs) within the NMC Controller as shown in Sect. 4.3 to ensure that only a single task is running on each processing entity.

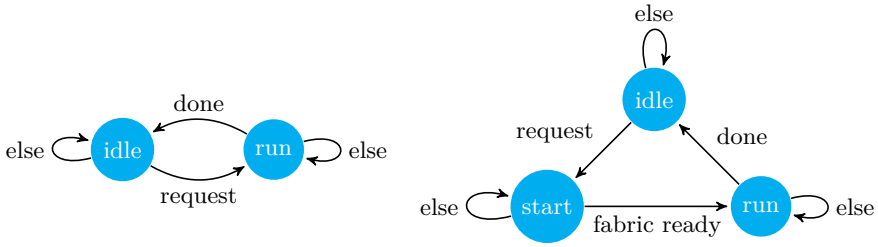

Fig. 6: FSMs of software-assisted (left) and direct hardware tasks (right)

4.4 Detailed Region Configuration

To allow clients to configure regions we implemented externally accessible registers of 32 bits within the NMC Controller. Table 1 shows the register set for a single region, which is available for each region configured at design time.

Offset	Register	Usage
0x00	Region Control Register	Operation selection, bus protection, SW/HW mode
0x04	Region Range Register	Start address and size
0x08	Region Status Register	Busy and finished bits
0x0A	Region Result Register	Result of the last operation
0x10	Region Operand Register	4x32 bits operand registers (HW mode only)

Table 1: Register set for a single region

The Region Range Register contains both start address and size of the memory region in the SPM. As operations are currently limited to 2^{16} words, it is sufficient to store both address and size in a single register to minimize hardware overhead.

Furthermore, the Region Control Register allows the user to enable a region and to control whether the region should not be readable during execution. It also includes operation field, which is either forwarded to the processor to identify the desired NMC operation or to the reconfigurable fabric to select the SI to be executed, depending on the selected operation mode.

The Region Status Register allows clients to check the execution state. Its busy bit is set as soon as the execution condition is met and cleared when the task has been finished. After the busy bit is cleared, clients can load the processed data from the SPM. Additionally, it is possible to retrieve the return value from the Region Result Register if necessary.

5 Evaluation

We implemented a prototype of the NMA using the reconfigurable processor *i*-Core [3], based on the LEON3 [1] core. This prototype is a dual-core system, where one core is the augmented near-memory processor and the second core is a standard LEON3 core acting as the remote user of the accelerator fabric. The resulting hardware design has been tested both in simulation and on a Xilinx VC707 prototyping board with a Xilinx Virtex 7 FPGA. Cycle counts have been determined using a clock cycle counter in the debug unit. During all tests, there was no additional traffic on the system bus by other applications. Such traffic would reduce the performance of software implementations even further, with low impact on the NMA performance as it has its own memory interface.

5.1 Video encoding acceleration

As part of a H.264 video encoder implementation on our platform, the sum of absolute differences (SAD) algorithm has been implemented within the NMA architecture [10]. It is used to calculate the similarity of two 16x16 image sections:

$$SAD(x, y, r, s) = \sum_{i=0}^{15} \sum_{j=0}^{15} |B_2(x + i, y + j) - B_1((x + r) + i, (y + s) + j)|$$

To support our approach, SAD is used as a benchmark to compare software-based execution to the NMA. In both cases, a remote processor configures a region and streams pixel data to that region. As soon as enough data is availabe for a SAD 16x16 operation, the NMC controller starts processing. In case of the accelerated implementation, the NMC Controller directly triggers the reconfigurable accelerator, not involving the processor at all. The software reference runs on the NMA processor, but does not use any accelerated instructions and is therefore limited to 32 bits bus width.

Table 2 shows a runtime improvement of 72% when launching the NMA through the remote processor interface. It also quantifies the memory interface

Interface	Software reference	near-memory accelerator	
		4 Pixels/Iteration	8 Pixels/Iteration
32 bits	1.635 cycles	463 cycles	339 cycles
128 bits	N/A	223 cycles	126 cycles

Table 2: Runtime comparison SAD 16×16

impact, taking only 48% resp. 37% cycles when using the high-bandwidth memory interface. Additionally, reconfiguration allows the SAD accelerator to be implemented twice, as enough resources are available in our sample design. Thus it can process eight pixels per iteration, almost doubling processing power.

5.2 AES encryption

To demonstrate both the flexibility and performance of the software-assisted approach, an AES encryption benchmark is used. The implementation is based on the *MiBench* suite [9]. As the AES algorithm involves a diverse control flow as well as preliminary operations like key derivation, the software-assisted operation mode is used. Consequently, a finished write to the input data region triggers an interrupt and preparations are done in software. The operations for a single encryption round are implemented in hardware, processing 128 bits of data per cycle. The hardware accelerator therefore called for each encryption round (10, 12 or 14, depending on key length) except for the last one. As a slightly different implementation is necessary in the last round, we fall back to the original software implementation. This already highlights a benefit of the processor-based approach: instead of having to implement the full AES algorithm in hardware, less demanding operations like preparation or finalization can still be handled in software. Porting an algorithm to the accelerator is easier, but still yields significant performance gains as can be seen in Fig. 7. An intermediate result is also included to distinguish memory bandwidth and hardware acceleration gains. In the intermediate case, the AES algorithm does not use hardware acceleration but still profits from increased memory performance of the NMA processor.

It can be seen that the software performance increases by 37% when using the high bandwidth memory interface only. Using the LUT-based hardware acceleration, execution time improves by approx. 53% compared to the software implementation. While this is a valuable improvement, it can't meet the performance of full hardware-based AES implementations. However, the hardware resources of the accelerator can be reconfigured and used for different workloads, trading performance against flexibility.

5.3 Resource usage

The resource usage of our implementation can be divided in the processor core, the reconfigurable accelerator and the NMC Controller. To allow comparing our approach to other implementations we also provide numbers for a common

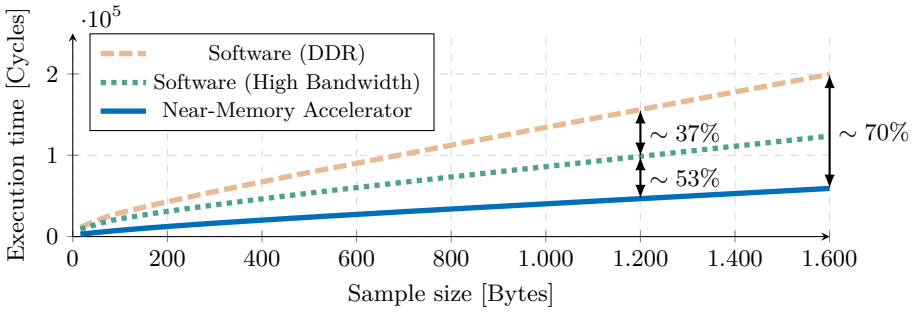

Fig. 7: AES performance comparison

LEON3 processor as well as a modified LEON3 core that can access a remote fabric within its execution stage. Table 3 shows FF, LUT and BRAM resources for the individual components.

	FF	LUT	BRAM
LEON3	4,785	12,793	22
Reconf. processor	5,505	13,245	24
fabric	7,922	16,326	59
instruction merging	1,032	9,661	0
Remote reconf. processor	5,471	13,741	24
NMC Controller (4 regions)	1,143	907	0

Table 3: FPGA resources by component

The reconfigurable fabric holds the biggest share of resources. This is primarily caused by the reconfigurable containers, holding a fixed amount of FPGA resources required to fit different accelerators. The amount of resources for these regions represents a tradeoff between low resource usage and flexibility to fit larger accelerators for applications, whose requirements may not be known at design time.

Figure 8 compares the resource requirements of the NMC Controller holding up to 32 regions to a configuration with N CPUs allowing to remotely access the reconfigurable accelerator using SIs. The difference between the resources allocated for a remote core with SI support and a common LEON3 processor yields the overhead necessary for remote execution. Our region-based approach uses standard LEON3 cores only, but requires to specify the amount of available regions at design time.

In terms of scalability, the resources used by the NMC Controller grows linearly with number of regions. Comparing to the original remote SI interface, the proposed concept is independent of the amount of processors in the system able to run instructions in the reconfigurable fabric. Instead, all clients share the

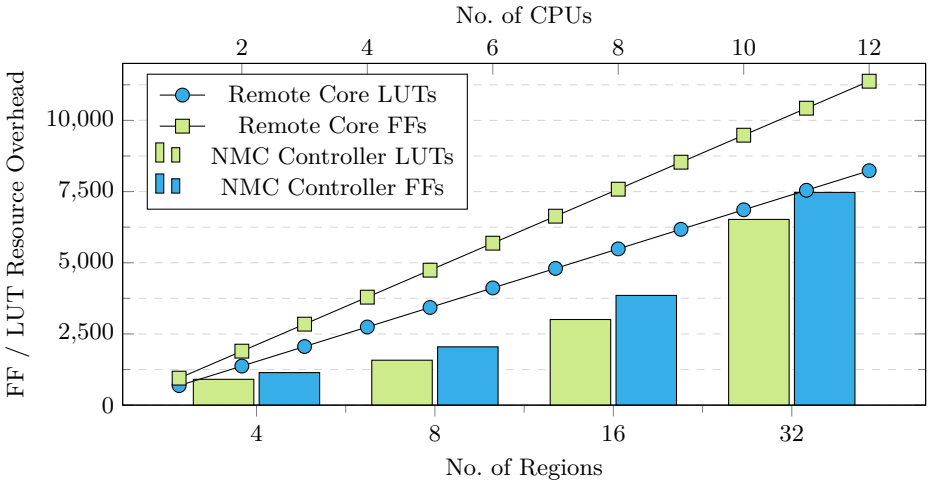

Fig. 8: Resource overhead comparison of remote SI cores and the region based approach

available regions and allocate regions on demand, allowing even more distant users in many-core environments (Section 3.5). The evaluation shows a tradeoff between processors with remote SIs support and the number of regions for the region-based approach, e.g. a six-core system exceeds both LUT and FF requirements of a design supporting 16 regions. However, the loose coupling between standard processors and a region-based NMA can lead to increased latency of workloads using a variety of accelerated operations, due to the overhead of configuring regions instead of calling an SI directly.

6 Conclusion

Throughout this paper, we present an approach for a highly flexible, programmable and reconfigurable near-memory accelerator. The concept gains high flexibility through a general-purpose processor, allowing to port existing implementations with low effort. The attached reconfigurable accelerator fabric can perform operations on the attached memory with high performance. It can be decided flexible if the accelerator is used directly or if it is sufficient to accelerate only parts of the algorithm and execute remaining tasks in software.

A core feature of our NMA is transparent operation on memory by defining regions with associated operations. The near-memory controller can dynamically execute these operations whenever a block of data is fully written to memory and notify the client after operation has finished. Especially in PGAS architectures, configuring and using the accelerator is greatly simplified, as buffers can be directly written to the address space of the client and no further IPC is necessary to start operation. For secure applications, the near-memory controller can lock access to data if required.

In future work, we will be looking into improving the control flow capabilities of the reconfigurable accelerator to depend less on the general-purpose processor and reducing its resource usage. Additionally, we want to investigate the impact of region-based NMC on large many-core platforms.

Acknowledgement

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center Invasive Computing [SFB/TR 89].

References

1. Aeroflex Gaisler AB: LEON3 Data Sheet (2010), `https://www.gaisler.com/doc/leon3_product_sheet.pdf`
2. Corda, S., Veenboer, B., Awan, A.J., Kumar, A., Jordans, R., Corporaal, H.: Near memory acceleration on high resolution radio astronomy imaging. In: 2020 9th Mediterranean Conference on Embedded Computing (MECO). pp. 1–6 (2020)
3. Damschen, M., Rapp, M., Bauer, L., Henkel, J.: i-Core: A runtime-reconfigurable processor platform for cyber-physical systems (01 2020)
4. Grudnitsky, A., Bauer, L., Henkel, J.: Corefab: Concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In: 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). pp. 1–10 (2014)
5. Gu, P., Xie, X., Ding, Y., Chen, G., Zhang, W., Niu, D., Xie, Y.: ipim: Programmable in-memory image processing accelerator using near-bank architecture. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 804–817 (2020)
6. Harbaum, T., Schade, C., Damschen, M., Tradowsky, C., Bauer, L., Henkel, J., Becker, J.: Auto-si: An adaptive reconfigurable processor with run-time loop detection and acceleration. In: 2017 30th IEEE International System-on-Chip Conference (SOCC). pp. 153–158 (2017)
7. Kang, W., Wang, H., Wang, Z., Zhang, Y., Zhao, W.: In-memory processing paradigm for bitwise logic operations in stt–mram. IEEE Transactions on Magnetics 53(11), 1–4 (2017)
8. Koo, G., Matam, K.K., I., T., Narra, H.V.K.G., Li, J., Tseng, H., Swanson, S., Annavaram, M.: Summarizer: Trading communication with computing near storage. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 219–231 (2017)
9. University of Michigan: MiBench Version 1.0 (2001), `https://vhosts.eecs.umich.edu/mibench/`
10. Wong, S., Vassiliadis, S., Cotofana, S.: A sum of absolute differences implementation in fpga hardware. In: Proceedings. 28th Euromicro Conference. pp. 183–188 (2002)

Some references removed for blind review.