An Energy-Efficient Integrated Programmable Array Accelerator and Compilation flow for Near-Sensor Ultralow Power Processing

An Energy-Efficient Integrated Programmable Array Accelerator and Compilation flow for Near-Sensor Ultra-low Power Processing

Satyajit Das*†, Kevin J. M. Martin*, Davide Rossi†, Philippe Coussy*, and Luca Benini†‡

*Univ. Bretagne-Sud, UMR 6285, Lab-STICC, F-56100 Lorient, France, [firstname].[lastname]@univ-ubs.fr
†Department of Electrical, Electronic and Information Engineering, University of Bologna, Italy, [firstname].[lastname]@unibo.it
‡Integrated Systems Laboratory, ETH Zurich, Switzerland, [first-initial][last name]@iis.ee.ethz.ch

***Abstract*—In this paper we give a fresh look to Coarse Grained Reconfigurable Arrays (CGRAs) as ultra-low power accelerators for near-sensor processing. We present a general-purpose Integrated Programmable-Array accelerator (IPA) exploiting a novel architecture, execution model, and compilation flow for application mapping that can handle kernels containing complex control flow, without the significant energy overhead incurred by state of the art predication approaches. To optimize the performance and energy efficiency, we explore the IPA architecture with special focus on shared memory access, with the help of the flexible compilation flow presented in this paper. We achieve a maximum energy gain of 2×, and performance gain of 1.33× and 1.8× compared with state of the art partial and full predication techniques, respectively. The proposed accelerator achieves an average energy efficiency of 1617 MOPS/mW operating at 100MHz, 0.6V in 28nm UTBB FD-SOI technology, over a wide range of near-sensor processing kernels, leading to an improvement up to 18×, with an average of 9.23× (as well as a speed-up up to 20.3×, with an average of 9.7×) compared to a core specialized for ultra-low power near-sensor processing.**

***Index Terms*—CDFG, CGRA, compilation, control flow, ultra-low power accelerator, computer architecture**

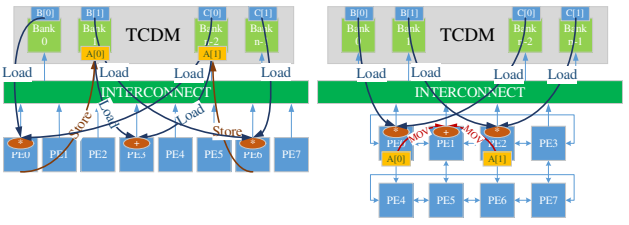
I. Introduction

Due to the increasing complexity of near-sensor data analytics algorithms, low power embedded applications such as Wireless Sensor Networks (WSN), Internet of Things (IoT), and wearable sensors combine the requirement of high performance and extreme energy efficiency in a mW power envelope [1]. While traditional ultra-low power sensor processing circuits rely on hardwired Application Specific Integrated Circuit (ASIC) architectures [11], near-threshold parallel computing is emerging as a promising solution [47]. Even though this approach provides maximum flexibility, a dominating majority of the power consumed during processing is linked to the typical overheads of instruction processors [15], such as instruction fetching and decoding, control and data-path pipeline overheads (up to 40%), load/store overhead (up to 30%). In this work, we make significant step forward in parallel near-threshold computing toward the goal of achieving the energy efficiency of application-specific data-paths, by exploiting the Coarse Grain Reconfigurable Array (CGRA) architectural template, and revisiting it to fit within an ultra-low power (mW) power envelope.

CGRAs have been intensely investigated in the past for applications with power consumption profiles ranging from mobile (hundreds of mW) [10] to high performance (hundreds of W) [34]. In this paper, we focus on a CGRA architecture in the mW range (and below). Very few CGRA architectures have been pushed in this ultra-low power mission profile [36] [48] [12]. Our CGRA is designed to work as an accelerator of an ultra-low power PULP processor cluster [47], sharing L1 memory with the processors. Hence, another major challenge in this context is achieving efficient L1 memory sharing [47]. To reduce memory access contention, it is necessary to have enough banks in the shared memory. On the other hand, the number of ports into the multi-banked shared-L1 memory logarithmic interconnect must be tightly constrained to avoid significant area and power overheads [44].

To cope with the ultra-low power profile and memory sharing challenges we build upon the Integrate Programmable-Array accelerator (IPA) concept proposed in [9] involving a multi-bank Tightly Coupled Data Memory (TCDM) coupled with a flexible and configurable memory hierarchy for data storage. As shown in Figure 1, from an architectural viewpoint, point-to-point data communication between processing elements (PEs) during kernel execution, represents a key advantage over energy-hungry data sharing over shared memory that is required when using a traditional processor-cluster architecture for parallel processing. The IPA cluster performs a lower number of memory operations on the sample program presented in the Figure 1(c), which in turn gives and energy improvement of 1.3× over the clustered multi-core architecture, which performs data sharing through the TCDM. In this comparison, we even ignore the barrier synchronization overheads in the many-core cluster for the sake of simplicity.

The IPA approach allows to significantly reduce the pressure on L1 memory, and hence the complexity of the interconnect between the PEs and the memory banks, since it requires a smaller number of banks to achieve low contention [47]. On the other hand, as opposed to clustered multi-core architectures, where data-exchange among cores is managed




```
for(i=0; i<1; i++)
{
A[i] = B[i] * C[i]
}
for(i=0; i<1; i++)
{
sum = sum + A[i];
}
```

(c)

Average energy consumption in pJ/operation

	Load-Store operations	Arithmetic operations	MOV operation		
	4.2	3.4	3.1		
	Total #LS	**Total #Arithmetic**	**Total #MOV**	**Energy (pJ)**	**Gain**
Many-Core	8	3	0	43.8	-
IPA	4	3	2	33.2	1.3x

(d)

Figure 1: (a) Cluster of processors; (b) Cluster with shared memory IPA accelerator (c) Sample program running in both the clusters; (d) Energy consumption comparison between the two clusters

through shared data structures and OpenMP parallel processing constructs, in CGRAs the compiler must take care of data-exchange among PEs, exploiting as much as possible point-to-point connections among PEs to minimize accesses to the shared memory.

Another major compiler challenge towards achieving high energy efficiency in CGRAs is the management of loop-carried data dependencies and control dependencies. State of the art compilers [37] [18] [40] [4] for CGRAs deal only with the straight-line code sequence (basic block) of the innermost loop of a kernel. In case of conditional present in the innermost loop, the compilers use predication [45] techniques to convert the control flows into data flow structures. Indeed, these compilers can only generate code to execute a single loop, as a set of pipelined stages is repeatedly executed up to a certain number ($loop\ boundary\ /\ number\ of\ pipelined\ stages$) specified by the compiler. In case of nested loops, only the innermost loop is accelerated using a CGRA, leaving the outer loops for the host processor. However, this approach requires several offloads by the host, which implies additional memory-mapped I/O operations for synchronization and communication with the CGRA. Hence, it causes significant overhead, especially when the innermost loop has a very small number of iterations, which is a typical scenario for near-sensor processing applications [52].

On the other hand, large CGRA architectures for high performance computing have frequently resorted to predication techniques to expose parallelism across control dependencies, such as conditionals [45] [19]. Unfortunately, predication leads to waste of resources and it is hard to justify in an extremely power and area constrained scenario [55]. In this paper, we address the above challenges by proposing a novel compilation flow tailored for our ultra-low power IPA architecture. This flow enables the execution of multiple loops and conditionals starting from ANSI C code, relying on the energy efficient *register allocation approach* presented in [7].

In a nutshell, this paper contributes to the two critical aspects of energy-efficient application mapping onto CGRAs. First, we carry out an architectural exploration, based on the IPA proposed in [9], for optimizing performance and energy efficiency. The IPA features full support for conditional operations, exploits the internal registers of the PEs for intermediate data exchange and relies on a multi-bank TCDM only for accesses to input/output buffers, significantly improving energy efficiency. Second, we describe a complete compilation flow to map kernels with multiple loop nests and conditionals onto the IPA. The flow helps releasing the host processor from performing the computation of the outer loops, significantly improving performance of the IPA. It also achieves high energy efficiency by minimizing the number of memory operations exploiting the features of the architecture.

To quantify the efficiency of IPA architecture and compilation flow, we compare the performance and energy consumption with the state of the art predication methods running on the IPA. Experimental results on a benchmark set of control intensive kernels show that the register allocation approach achieves a maximum of 1.33× (with minimum of 1.04× and average of 1.13×) and 1.8× (with minimum of 1.37× and average of 1.59×) performance gain compared to partial predication and full predication techniques. For what concerns shared-L1 memory access optimization, our exploration shows that a banking factor of 0.5 (i.e. 8 LSUs, 4 TCDM banks) provides the optimal configuration in terms of performance and energy for a IPA configuration with 4x4 PEs. Moreover, the IPA features a very regular control and data-path structure, which is suitable for fine-grained power management. We exploit this architectural regularity to design a fine-grained clock gating mechanism, which turns into an average 2× energy efficiency boost with respect to the non-clock-gated implementation of the IPA. Results show that the IPA achieves a maximum speed up of 20.3×, with an average of 9.7× compared to one or1k processor [16][1], with an area ratio of just 1.6×. The average energy efficiency achieved by the IPA operating at 0.6V is 1617 MOPS/mW, which is up to 18× and on average 9.23× better than what is achieved by the processor.

The rest of this paper is organized as follows. In Section II, the background and related work are discussed. In section III, the target architecture, memory hierarchy and the execution model are described. Section IV focuses on presenting the full compilation flow, with the support of required definitions, and models. Section V presents the implementation and experimental results. Finally, the paper is concluded in Section VI.

II. Background and Related Work

Much research has been done to evaluate the performance, power, and cost of CGRA architectures [10]. In this paper, we focus on the energy efficiency aspects of both the architecture and the compiler.

A. Architecture

While targeting low power execution, data and context management is of utmost importance, integration of CGRAs

[1]This processor is optimized for low power execution in the context of near threshold near-sensor processing

as accelerators with the data and instruction memory has seen several solutions over the past years [10].

In many low-power targeted CGRAs [2][39][49][23], memory operations are managed by the host processor. Among these architectures, Ultra-Low-Power Samsung Reconfigurable Processor (ULP-SRP) and Cool Mega Array (CMA) operate in ultra-low-power (up to 3 mW) range. In these architectures, PEs can only access the data once prearranged in the shared register file by the processor. For an energy efficient implementation, the main challenge for these designs is to balance the performance of the data distribution managed by the CPU, and the computation in the PE array. However, in several cases, the computational performance of the PE array is compromised by the CPU, due to large synchronization overheads. For example, in ADRES [2] the power overhead of the VLIW processor used to handle the data memory access is up to 20%. In CMA [39] the host CPU feeds the data into the PEs through a shared fetch register (FR) file. This is very inefficient in terms of flexibility. The key feature of this architecture is the possibility to apply independent DVFS [53] or body biasing [36] to balance array and controlling processor parameters to adjust performance and bandwidth requirements of the applications. The highest reported energy efficiency for CMA is 743 MOPS/mW on 8-bit kernels, not considering the overhead of the controlling processor, which is not reported. With respect to this work, which only deals with DFG described with a customized language, we target 32-bit data and application kernels described in C language, which are mapped onto the array using an end-to-end C-to-CGRA compilation flow.

In architectures such as, MorphoSys [51], RSPA[24], SmartCell [29], PipeRench [17], SIMD-CGRA [14], load-store operations are managed explicitly by the PEs. Data elements in these architectures are stored in a shared memory with one memory port per PE row. The main disadvantages of such data access architecture are: (a) lots of contention between the PEs on the same row to access the memory banks, and (b) expensive data exchange between rows through complex interconnect networks within the array. With respect to these architectures, our approach minimizes contention by exploiting a multi-banked shared memory with word-level interleaving. In this way data-exchange among tiles can be performed either through the much simpler point-to-point communication infrastructure or fully flexible shared TCDM.

Solutions targeting high programmability and performance executing full control and data flows are reported for the weakly programmable processor array (WPPA) [25], Asynchronous Array of Simple Processors (AsAP) [56], RAW [54], ReMAP [6], XPP [3], and TRIPS [50]. ~~The WPPA array consists of VLIW processors, and for low power target, the instruction set of a single PE is minimized according to domain-specific computational needs. In AsAP, each processor contains local data and instruction memory, FIFOs for tile-to-tile communication, and local oscillator for local clock generation. Both the ReMAP and XPP consist of PE array each with DSP extension. These architectures are mainly intended for exploitation of task-level parallelism, and each processor of the array must be programmed independently. RAW PEs consist of 96 KB instruction cache and 32 KB data cache, router based communication.~~ These large-scale "array of processors" CGRAs are out of scope for ultra-low power, mW-level acceleration. Still there are reported in Table IX for comparison.

NASA's Reconfigurable Data-Path Processor (RDPP) [12], and Field Programmable Processor Array (FPPA) [13] are targeted for low-power stream data processing for spacecrafts. These architectures rely on control switching [12] of data streams, and synchronous data flow computational model avoiding investment on memories and control. On the contrary, the IPA is tailored to achieve energy-efficient near sensor processing of data with workloads very different from the stream data processing.

Table I summarizes an overview of the jobs managed by CGRA and the host processor for different architectural approaches. Acceleration of the kernels involves memory operations, innermost loop computation, outer loop computation, offload and synchronization with the CPU. As shown in the table, IPA manages to execute both the innermost and outer loops, and the memory operations of a kernel imposing least communication and memory operation overhead while synchronizing with the CPU execution.

With respect to these state of the art reconfigurable arrays and array of processors, this paper introduces a highly energy efficient, general-purpose IPA accelerator where PEs have random access to the local memory, and execute full control and data flow of kernels on the array starting from a generic ANSI C representation of applications [7]. This paper also focuses on the architectural exploration of the proposed IPA accelerator [9], with the goal to determine the optimal configuration of number of LSUs and number of banks for the shared L1 memory. Moreover, we employ a fine-grained power management architecture to eliminate dynamic power consumption of idle tiles during kernels execution which provides 2× improvement of energy efficiency, on average. The globally synchronized execution model, low cost but full-flexible programmability, tightly coupled data memory organization, and fine-grained power management architecture define the suitability of the proposed architecture as an accelerator for ultra-low power embedded computing platforms.

B. Compilation

To map the loops, state of the art compilers for CGRA mostly rely on software pipelining [18] [37] [40]. This approach can manage to map the innermost loop body in a pipelined manner. On the other hand, for the outer loops, CPU must initiate each iteration in the CGRA, which causes

Table I: Comparison between different architectural approaches

References	[2][39][48] [13][12] [38][35]	[31]	[51][24] [17][5]	IPA This paper
Memory ops	CPU	CGRA	CPU	CGRA
Innermost loop	CGRA	CGRA	CGRA	CGRA
Outer loop	CPU	CPU	CGRA	CGRA
Offload + Sync	CPU	CPU	CPU	CPU
Overhead				

Table II: Comparison between different approaches to manage control flow in CGRA

Techniques	Conditionals		Loops	
	Balanced	Imbalanced	Single	Nested
Partial predication [19]	√	√	×	×
Full predication [20]	√	√	×	×
State based full predication [21]	√	√	×	×
Dual issue single execution [19]	√	×	×	×
TLIA [32]	√	√	√	×
Software pipelining [37]	×	×	√	×
Loop unrolling [27]	×	×	√	NA
Register allocation [7]	√	√	√	√

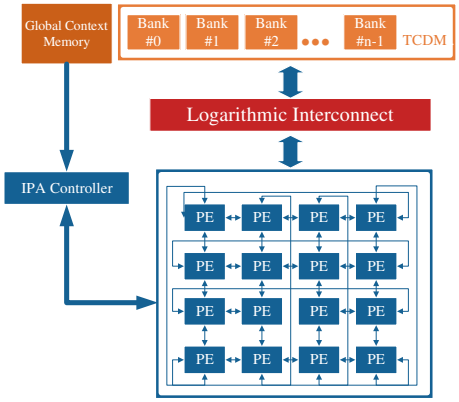


Figure 2: Integrated Programmable-Array Accelerator

significant overhead in the synchronization between the CGRA and CPU execution. Liu et al in [31] pinpointed this issue and proposed to map maximum of two levels of loops using polyhedral transformation on the loops. However, this approach is not generic as it cannot scale to an arbitrary number of loops. Some approaches [30] [27] use loop unrolling for the kernels. The basic assumption for these implementations is that the innermost loops trip count is not large. Hence, the solutions end up doing partial unroll of the innermost loops. The outer loops remain to be executed by the host processor. As most of the proposed compilers handle innermost loop of the kernels, they mostly bank upon the partial predication [19] [37] and full predication [20] techniques to map the conditionals inside the loop body.

Partial predication maps instructions of both if-part and else-part on different PEs. If both the if-part and the else-part update the same variable, the result is computed by selecting the output from the path that must have been executed based on the evaluation of the branch condition. This technique increases the utilization of the PEs, at the cost of higher energy consumption due to execution of both paths in a conditional. Unlike partial predication, in full predication all instructions are predicated. Instructions on each path of a control flow, which are sequentially configured onto PEs, will be executed if the predicate value of the instruction is similar with the flag in the PEs. Hence, the instructions in the false path do not get executed. The sequential arrangement of the paths degrades the latency and energy efficiency of this technique.

Full predication is upgraded in state based full predication [21]. This scheme prevents the wasted instruction issues from false conditional path by introducing sleep and awake mechanisms, but fails to improve performance. Dual issue scheme [19] targets energy efficiency by issuing two instructions to a PE simultaneously, one from the if-path, another from the else-path. In this mechanism, the latency remains similar to that of the partial predication with improved energy efficiency. However, this approach is too restrictive, as far as imbalanced and nested conditionals are concerned. To map nested, imbalanced conditionals and single loop onto CGRA, the triggered long instruction set architecture (TLIA) is presented in [32]. This approach merges all the conditions present in kernels into triggered instructions, and creates instruction pool for each triggered instruction. As the depth of the nested conditionals increases the performance of this approach decreases. As far as the loop nests are concerned, the TLIA approach reaches bottleneck to accommodate the large set of triggered instructions into the limited set of PEs.

The compilation flow we propose, uses the register allocation approach [7] to map CDFGs onto the CGRA. This allows to map both loops and conditionals of any depth. In our case, the only limitation in the mapping of kernels onto the CGRA is given by the size of instruction memory of the PEs, and not by the structure of the application (i.e. number of loops, and branches). Also, one can increase the size of code segment to be executed in the CGRA as much as possible, minimizing the control and synchronization overheads with the core, which is not negligible in the other approaches. Table II presents a comprehensive comparison between several techniques to manage control flow in the kernels. Software pipelining and loop unrolling are mostly used for the mapping of the innermost loop, while branches inside the loop are managed by one of the described predication techniques. Hence, the existing compilers use combined solutions for branches and innermost loop mapping. This requires exhaustive exploration to find out the most suitable combination for the target architecture and application domain. On the contrary, our proposed compilation flow ~~uses a register allocation approach which~~ can handle both conditionals and loops efficiently.

III. IPA Architecture and Execution Model

In this section, we present the general-purpose Integrated Programmable-Array Accelerator (IPA) architecture, supporting standalone execution of complete control and data flow applications.

A. Integrated Programmable-Array Accelerator (IPA)

IPA is the integration of a PE Array (PEA) and a tightly coupled data memory (TCDM) through a low-latency logarithmic interconnect. An IPA controller loads the context into the PEs from a pre-loaded Global Context Memory (GCM). Figure 2 shows the organization of the IPA.

The PEA consists of a parametric number of PEs connected with a 2-dimensional tours network. The PE Array follows the multiple instruction, multiple data (MIMD) model of computation. All PEs operate on different set of instructions. A bus based interconnect network is implemented to load instructions

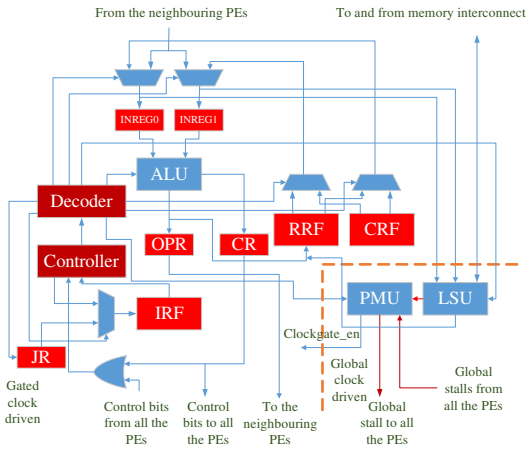


Figure 3: Components of the PE

and constants (i.e. context) from the GCM into the PEs, whereas the torus network is used during execution phase for low power data communication between the PEs. The details of the load context protocol are discussed in [9]. To achieve low power execution, the instruction set architecture [9] was designed from scratch resulting 20-bit long instruction. We took the advantage of the visibility of the micro-architecture to the compiler and shifted the immediate data to constant register file in the PEs (discussed later) which eases the compression of the instruction, imposing low pressure on the decoder.

Figure 3 describes the components of a PE. The Load Store Unit (LSU) is optional for the PEs (the optimal number of LSU is a parameter studied in this paper). Two operands (IN0 and IN1) define the inputs of each PE. The input sources are the neighbouring PEs and the register file. A 32-bit ALU and a 16-bit $\times$ 16-bit $\rightarrow$ 32-bit multiplier are employed in this block. The Constant Register File (CRF) stores the immediate values of the instructions, while the Regular Register File (RRF) and Output Register (OPR) store the temporary variables. The Controller fetches the instructions from the Instruction Register File (IRF). If the decoded instruction is a *jump*, the target address of the *jump* is stored in the Jump Register (JR). The *cjump* (conditional jump) instruction contains two target addresses. The true path is evaluated in the JR by the Boolean "OR" of the Condition Register (CR) bits of the PEs.

The TCDM acts as L1 memory for the IPA. Featuring a number of ports equal to the number of memory banks, the TCDM provides concurrent access to different memory locations. The TCDM is interfaced with the LSUs of the PE array through a low latency, logarithmic interconnect [44], implementing a word level interleaving scheme to minimize access contention.

B. Power Management Unit (PMU)

To reduce dynamic power consumption in idle mode, each PE contains a tiny Power Management Unit (PMU) which clock gates the PEs when idle. An idle condition for a PE arises from three situations: (i) Unused PE: when a PE is not used during mapping; (ii) Load Store stall: In case of TCDM banking conflict the PMU generates a *global stall*, which is broadcast to all the PEs. Until the global stall is resolved, all the PEs are clock gated by their corresponding PMUs. LSUs are placed in the global clock region (Figure 3) to avoid deadlocks; (iii) Multiple NOP operations: a NOP instruction contains the number of successive NOPs. When a NOP instruction is fetched, the decoder loads this number into a counter within the PMU. The *clockgate_en* remains low until the count reaches zero. The counter gets halted when it encounters a global stall and resumes the count after the stall is resolved, synchronizing the execution flow among PEs.

Due to the fine-grained nature of the power management, more aggressive power gating is not implemented, since it imposes large area penalty without significant benefits. The leakage power of each tile is so small that it does not change significantly the energy efficiency when the rest of the system is active.

Table III: Instruction format

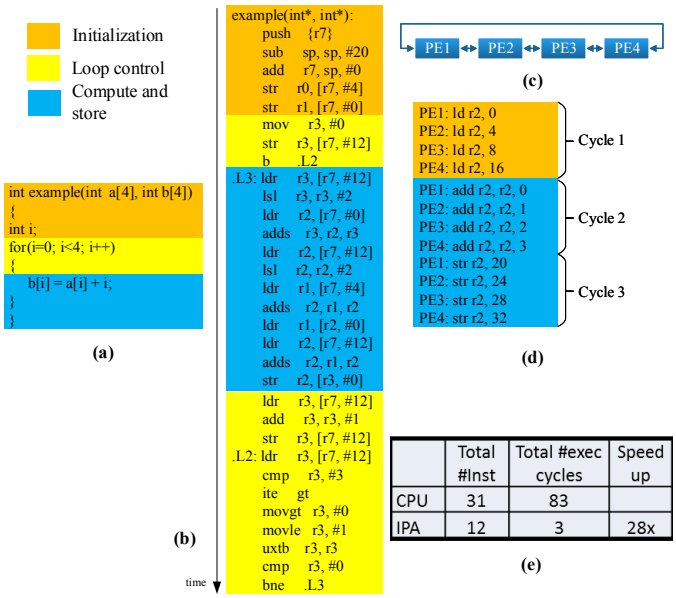
5 bits	2 bits	3 bits	1 bits	4 bits	1 bit	4 bits
Opcode	Output Reg type	Dest Reg Addr	IN0 Type	IN0 Addr	IN1 Type	IN1 Addr
Jmp	Address		unused			
Cjmp	Address of the true path		Address of the false path		unused	
NOP	Number of consecutive NOPs		unused			

C. Overview of the execution model

After compiling a kernel (see section IV), the compiler generates the assembly and the addresses for the input and output data in the local shared memory. The assembler takes the assembly and the Instruction Set Architecture (ISA) of the IPA, to generate the context (i.e. the program to be stored into the IRF) for each PE, which is pre-loaded in the GCM. The context contains instructions and constants for each PE in the array. Prior to the execution start, the context is loaded into the corresponding IRF and CRF of the PEs. We assume that the code fits in the local memory. Larger execution contexts can be handled using the IPA controller and overlays. Details on this process are omitted for the sake of conciseness[2]. In each cycle, the PEs fetch 20-bit instruction from the local IRF. The immediate data are shifted to constant register file which eases the compression of the instruction. Hence, the pressure on the decoder is quite low. Table III describes the instruction format.

Figure 4 shows the execution of a sample program in a traditional CPU and the IPA. The total number of instructions for the sample program in the CPU and the IPA are 31 and 12 respectively. Also, the IPA achieves 28$\times$ performance gain compared to that of the CPU while executing the sample program. The decrease in the number of instructions in the IPA in this specific example is mainly due to the much lower number of memory operations and the fact that the small loop can be completely unrolled without code size blown-up.

[2]Note that the context loading and setup cost are accounted for in the experimental results.



	Total #Inst	Total #exec cycles	Speed up
CPU	31	83	
IPA	12	3	28x

Figure 4: (a) Sample program (b) Execution in CPU (c) Example PEA (d) Execution in IPA (e) Execution metrics in CPU and IPA

IV. Compilation Flow

The compilation generates a mapping of the program for the corresponding PEA. In this section, we present the models adopted for the architecture and the application and the full compilation flow to map control and data flow onto the PEA. We also discuss the register allocation approach to exploit the register files of the PEs while preserving control-carried dependencies.

A. Architecture, application model and homomorphism

The compiler takes two inputs. The first is the PEA model, and the second is the ANSI-C code of the application.

The PEA is modelled by a bipartite directed graph with two types of nodes: operators and registers. Timing is implicitly represented by connections between registers and operators, which is referred to as the *time extended model* of the PEA [18]. Two types of operator nodes are defined for the PEAs. The first type is the computing operator (functional unit (FU) nodes in Figure 5(a)) that represents the physical implementation of an arithmetic and logical operation (+, ×, -, OR, AND) and/or memory access (e.g. load/store). The second type of operator is the memorization operator (circular nodes in Figure 5(b)). It is associated with the output register and represents the operation of keeping a value in a local register explicitly.

Figure 5 (a) shows a sample PEA with two PEs connected by a torus network. Each PE has 3 registers in the distributed register file, and a single output register. Figure 5 (b) represents the *time extended model* of the PEA shown in Figure 5 (a). In this model, one can vary the interconnect network, the distribution and size of the register file, and the type of the PE, to explore different PEA architectures.

The application is modelled as a control and data flow graph (CDFG). Supporting control flow gives the opportunity

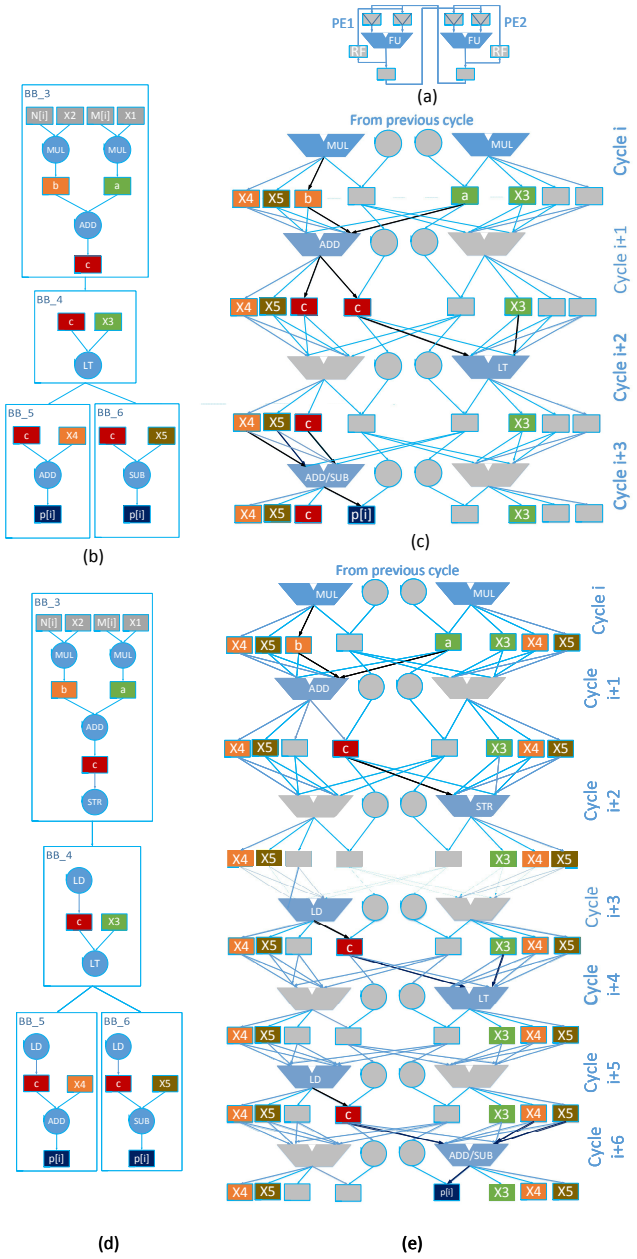


Figure 5: (a) A 2×1 PEA with 3 registers in RF and one output register (b) CDFG model (c) A possible mapping of (b) onto the PEA over 4 cycles using register allocation based approach. (d) The transformed CDFG of (b) for systematic load store based approach (e) A possible mapping of (d) onto the PEA over 7 cycles using systematic load store based approach

to accelerate a kernel without any intervention of the host processor. A CDFG is depicted as $G = (V, E)$ where V is the set of basic blocks and $E \subseteq V \times V$ is the set of directed edges representing control flow. A Basic Block (BB) is represented as a data flow graph (DFG) or $BB = (D, O, A)$ where D is the set of data nodes, O is the set of operation nodes and A is the set of arcs representing dependencies. The control flow from one basic block to another is supported with jump (*jmp*) and conditional jump (*cjmp*) instructions.

Fig. 6 presents a sample program and the corresponding CDFG. In this figure, basic blocks are represented as blue

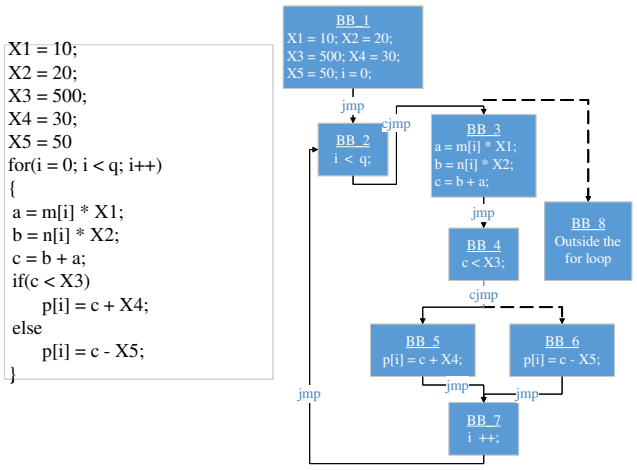


Figure 6: Sample program and corresponding CDFG

rectangles. The flow from one basic block to another basic block is represented by black arrows and managed by simple branch (*jmp*) operation. The true and false paths of a conditional managed by *cjmp*, are shown by solid and dashed arrows respectively. The execution flow of the CDFG is presented as: $BB\_1 \rightarrow BB\_2 \rightarrow$ (either BB_3 or BB_8) if $BB_3 \rightarrow BB\_4 \rightarrow$ (either BB_5 or BB_6) $\rightarrow BB\_7 \rightarrow BB\_2 \cdots$. In order to maintain the execution flow, it is necessary to synchronize all the PEs in the array, to the execution of the same basic block. When the execution flow jumps from one basic block to another, all the PEs in the PEA must be synchronized to the current basic block execution. This allows to use all the PEs concurrently or sequentially, while executing a single basic block. Dually, several basic blocks can use the same PE. The synchronized execution allows the compiler to map several operations and data onto the same PE. Next, we present the homomorphism of the CDFG model with the application model, to support different stages in the compilation flow.

The basic blocks in the CDFG, presented in Figure 5(c), are composed of data nodes, operation nodes, and data dependencies. Three equivalences between the basic block DFGs and PEA model nodes are defined: (1) data and registers; (2) computation and computing operators; (3) data dependences and connection between the time extended PE components. As the two models are homomorphic, the mapping of a DFG onto the PEA is therefore a problem equivalent to finding a DFG in the PEA graph.

Figure 5(b) represents a possible mapping of the sample CDFG in Figure 5(c) onto the PEA in Figure 5(a) over 4 cycles. In the next section, we discuss the full compilation flow for CDFG mapping.

B. The compilation flow step by step

Figure 7 shows a schematic representation of the compilation flow for mapping CDFGs onto the PEA. A CDFG mapping is a set of DFG mappings that are compatible with each other. To be compatible, the DFGs must access the data that remain in the PEs (see symbol variables (see definition IV.1)) in the same location. This is ensured by the register allocation approach.

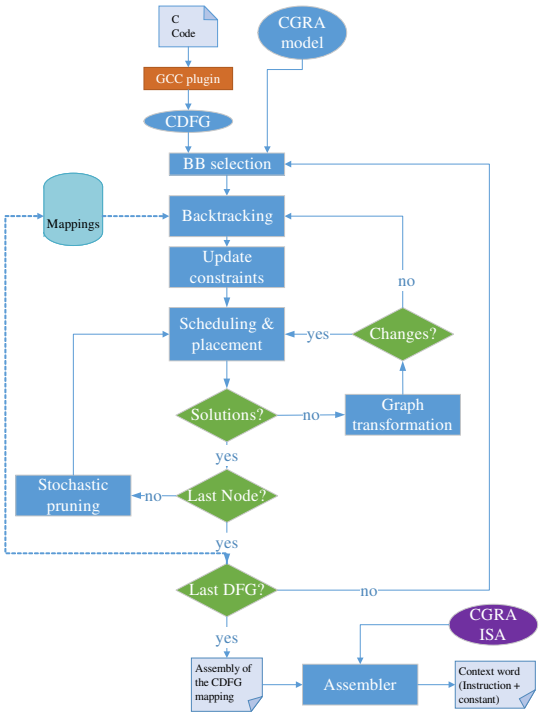


Figure 7: Compilation flow

To map the basic blocks, we rely on the highly scalable and efficient mapping approach for DFGs described in [8]. The compilation flow proposed in this paper, extends the DFG mapping to accommodate the register allocation approach to map a full CDFG onto the PE array. As presented in Figure 7, the full compilation flow is composed of six interdependent stages: BB selection, backtracking, update constraints, scheduling and placement, graph transformation and a stochastic pruning. These tasks are described in detail in the next sub-sections.

1) Scheduling and placement: The proposed approach uses a backward traversal [43] list scheduling algorithm to schedule the DFG of each basic block. It relies on a heuristic in which the schedulable operations are listed by priority order. In backward traversal, a node is schedulable if and only if all its children are already scheduled. The priority of nodes depends on their mobility [42]. It is possible to process memorization nodes and conventional nodes differently. Also, when several nodes have the same mobility, their respective number of successors is used as a second priority criterion. The higher is the number of successors, the higher the priority is. Indeed, a node with a higher number of successors is more difficult to map due to the routing constraint coming from the limited amount of connections between tiles. Thus, scheduling these nodes at first usually allows to decrease the application's latency [43]. As soon as the highest priority node has been defined, the compiler tries to find a placement in the PE array model. If a placement solution exists, the node is scheduled else the graph is transformed.

The proposed placement uses an incremental version of Levi's algorithm [28]. The proposed algorithm adds the newly scheduled operation node and its associated data node to the sub-graph composed of already scheduled and placed nodes.

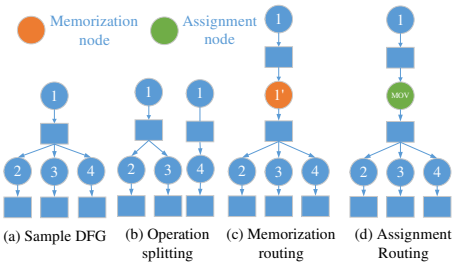


Figure 8: Graph transformation

Only the previous set of solutions that have been kept, location constraints (RLC (see definition IV.3) and TLC (see definition IV.2)) are used to find every possibility to add this couple of nodes without considering the non-yet scheduled nodes. If no solution is found, there is absolutely no possibility to bind this couple in all the previous partial solutions because Levi's algorithm provides a complete exploration of the solution space. In that case, graph transformation is required.

2) Graph transformation: DFG is transformed dynamically when no binding solution is found. The three graph transformations are used in our compilation flow (Figure 8).

- *Operation splitting* duplicates an operation node by keeping its same inputs and distributing output edges to reduce the number of successors of the original operation node.
- *Memorization routing* adds a memorization node and its associated data node to delay one operation and to keep data dependencies
- *Assignment routing* adds an assignment node (*mov* operation node) to increase the physical distance between the source and sink of symbol variables by one. Due to TLC or RLC, when the physical distance between the source and sink of the symbol variable becomes more than one, the compiler dynamically adds one *mov* operation node to the DFG.

3) Stochastic pruning: The exactness of the placement approach leads to very large number of partial mappings. And it grows exponentially if not pruned. Hence, we use the stochastic pruning approach described in [8].

4) Basic block selection: Once all the nodes of the BB have been scheduled and bound, the compiler selects one mapping among the several mappings generated, and selects the next basic block to be mapped. As discussed previously, it is necessary to maintain data integrity over several basic block mappings. The data mapping problem for CDFG mapping is now described before going into the detailed basic block selection step.

4.a: Definition and problem formulation

Data in an application is separated into two categories. (i) The standard input and output data (mostly the array inputs and outputs) are mapped as memory operands. The inputs and outputs are allotted by load-store operations. In our sample program in Figure 6, m, n are the input arrays and p is the output array, which are managed by load and store operations. (ii) The internal variables of a program are mapped onto the registers of the processing elements, and managed by the register allocation based approach [7]. Following, we introduce several definitions concerning register allocation approach:

Definition IV.1. [Symbol Variables and location constraints] In compilation, the recurring variables (repeatedly written and read) are managed in local register files of the PEs to avoid multiple access of local memory. The recurring variables which have occurrences in multiple basic blocks need special attention since the integrity of these variables must be kept intact throughout the mapping process for different basic blocks. These variables are defined as *Symbol variables*. The register locations for the symbol variables are referred to as *location constraints*. For instance, variable c in the CDFG (Fig. 6) is written in BB_3, and read in BB_4, BB_5 and BB_6. The register location for c must be same for all the mappings of these basic blocks. Similarly, $X1$, $X2$, $X3$, $X4$, $X5$, i, a and b must be location constrained. In the rest of the paper, the locations for such *symbol variables* are denoted with an overline, as $\overline{variable\_name}$.

Depending on the order of the basic blocks mapped (i.e. traversing the CDFG), some location constrains may be reused in the mapping process or may be kept reserved for later use. These two types of location constraints are discussed in the following.

Definition IV.2. [Target Location Constraints (TLC)] We consider a scenario *scenario_1*, where BB_6 is mapped first, BB_3 is mapped next and so on. While mapping BB_6, variables c and $X5$ are placed at $\overline{c}$ and $\overline{X5}$. While mapping BB_3, $\overline{c}$ and $\overline{X5}$ which are already mapped in BB_6, must be considered because $\overline{c}$ will be used to map c in BB_3. In other words, the placement of the variables in the registers must be respected. Also, $\overline{a}$, $\overline{b}$, $\overline{X1}$ and $\overline{X2}$ must not reuse $\overline{X5}$. Otherwise, $X5$ will have wrong value when executing BB_6. Let's consider *scenario_2* with another order of basic blocks mapped, like first BB_3 and then BB_6 and so on. In this order of mapping, it is necessary to pass $\overline{c}$ and $\overline{X5}$ from BB_3 to BB_6 mapping. To keep c and $X5$ alive in BB_6 both $\overline{c}$ and $\overline{X5}$ must be used in mapping of BB_6. The placement or binding information which are passed from the mapping of one basic block to the mapping of the other basic block is referred to as *constraint* (e.g. *scenario_1*: $\overline{c}$ and $\overline{X5}$ passed from BB_6 to BB_3). The location constraints related to the data that are used within a basic block mapping phase (e.g. *scenario_1*: $\overline{c}$ in BB_3 mapping) are referred to as *target location constraints* (TLC).

Definition IV.3. [Reserved Location Constraints (RLC)] As we have seen in the previous examples, some of the location constraints must be reserved in the mapping of basic blocks for the sake of data integrity. To keep the symbol variables alive, it is necessary to exclude the memory elements from placement. Accordingly, these resources will not override while mapping the basic block (e.g. *scenario_1*: $\overline{X5}$ in BB_3 mapping). These are referred to as *reserved location constraints* (RLC).

4.b: Selection approach

If the number of RLC and TLC is high, mapping becomes complex. As TLC will force to use resources, and RLC will force to exclude resources from placement. Hence, the primary goal for our compiler is to minimize the number of TLCs and RLCs by choosing an efficient traversal of the CDFG.

The basic solution to deal with the symbol variables is to

Table IV: Comparison of RLC and TLC numbers between different CDFG traversal

Kernels	Forward Traversal				Backward Traversal			
	Breadth First Search		DepthFirst Search		Breadth First Search		DepthFirst Search	
	# RLC	# TLC	# RLC	# TLC	# RLC	# TLC	# RLC	# TLC
Seperable 2D Filter	22	35	17	35	22	35	17	35
sobel Filter	64	85	35	85	69	85	35	85

introduce memory operations. The symbol variables are stored in the memory where they are written and are loaded from the memory when read. In the rest of the paper this basic solution is referred to as *systematic load-store based approach*. This method is presented in the Figure 5(d). For the symbol variable c in the CDFG shown in Figure 5(c), it stores variable c in the memory in BB_3, and loads in BB_4, BB_5 and BB_6. Figure 5 refers to the mapping of the transformed CDFG in this approach. This basic solution reduces the complexity of the mapping as there are no constraints to be dealt with while mapping the basic block. However, it requires a huge memory bandwidth, significantly reducing the energy efficiency of the system. As the compilation is built on *register allocation approach*, the symbol variables are stored in the register files when they are produced, and retrieved from the registers when used as operands. While doing so, the effects of the constraints in mapping are unavoidable. RLC restrict the use of some resources, and TLC force to reuse some resources. If there is only a single TLC in a basic block mapping, it becomes easier to start mapping from the known place. But several TLC and RLC complicates the mapping. Forced and blocked placements by these constraints induce extra routing effort (dynamically transforming the graph in compilation).

As the selection of the basic blocks during the mapping is important, we compare the number of TLC and RLC for several CDFG traversal in this section. Table IV presents the comparison between the number of different constraints in the forward and backward CDFG traversal for Breadth First Search (BFS) and Depth First Search (DFS) strategies. As the trend is similar for other kernels we present the results for sobel and seperable 2D filter only. The numbers show that DFS strategy generates a lower number of RLC than the BFS in both forward and backward traversal. The number of RLC for sobel filter is much higher in BFS due to several sequential loops present in the kernel. The numbers of TLC are similar in both the strategies for different traversal mechanisms. Also for the different search strategies forward and backward traversal perform similarly. The DFS strategy is thus used.

5) Backtracking: For a basic block to be mapped (except the first one), this stage selects the first map out of several mappings generated for the last basic block mapped. The selected map updates the constraints for the current basic block mapping. If one basic block does not find a mapping due to the constraints, this stage selects the second map from previous basic block to update the constraints and restart mapping of the new basic block. The process continues up to the first basic block mapped until a valid mapping is found for the current basic block.

6) Update Constraints: In this stage, the compiler creates and updates a constraint database. This database is used in the placement algorithm, to place the data nodes and corresponding operation nodes according to the TLC and RLC. When mapping a current basic block, new variables cannot be placed in RLCs, while TLCs are used to map the symbol variables. If the symbol variable in the current basic block mapping is not present in the constraint database, then the variable is mapped using available resources, and the respective placement is used to update the constraint database prior to next basic block mapping. Once all the basic blocks are mapped the compiler generates the assembly file containing a single map for the whole CDFG.

C. Assembler

Assembler holds the key to differentiate from the PEA model used in the compiler and the actual hardware implementation. The assembler takes the ASCII text assembly generated by the compiler and the instruction set architecture (ISA) and produces machine code, which can then be used to configure the PEs in the hardware. The ISA provides the added hardware information to the PEA model used in the compiler. As an example, the PEs in the IPA use an added constant register file (CRF) for storing the constants. The introduction of the CRF in the PEA model minimizes the instruction length by storing the immediates of the instruction into the internal registers, giving a low power solution. That is how the assembler separates the model used in the compiler from the actual implementation of the hardware. One can define their own PEA model and derive an architecture from that for actual implementation. Thus, the compiler can be used for a wide range of PEA variations.

V. Experimental Results

This section analyses the implementation results, providing performance, area, and energy consumption on several signal processing kernels. We carry out experiments to show the efficiency of the register allocation approach compared to the state of the art predication techniques, considering a wide range of control dominated kernels. We also perform an architectural exploration to find the optimal configuration in terms of number of load-store units and number of TCDM banks for a IPA with 4x4 PE array. Performance, area and energy efficiency are also compared with that of the or1k CPU [26].

A. Implementation Results

This section describes the implementation results for the IPA accelerator, providing a comparison with the or1k CPU. Both the designs were synthesized with Synopsys design compiler 2014.09-SP4 using STMicroelectronics 28nm UTBB FD-SOI technology libraries. Synopsys PrimePower 2013.12-SP3 was used for timing and power analysis at the supply of 0.6V, 25°C temperature, in typical process conditions. The cycle information was achieved simulating the RTL with Mentor Questa Sim-64 10.5c. In the following, the exploration

Table V: Code size and the maximum depth of loop nests for the different kernels in the IPA

Kernels	**FIR**	**MatM**	**Conv**	**Sep Filter**	**Non Sep Filter**	**FFT**	**DC Filter**	**cordic**	**sobel**	**gcd**	**sad**	**deblock**	**manh-dist**
Code size (KB)	0.568	0.704	0.704	0.720	0.784	0.696	1.16	0.496	0.336	1.448	0.600	2.016	0.624
Max depth loop nests	2	3	3	3	4	2	2	1	1	1	2	3	2

Table VI: Specifications of memories used in TCDM and each PE of the IPA

Name	Type	Size
Global context memory	SRAM	8KB
TCDM	SRAM	32KB
Instruction Register File (IRF)	Registers	0.08KB
Regular register file (RRF)	Registers	0.032KB
Constant register fie(CRF)	Registers	0.128KB

considers a 4×4 array with 16 PEs, each one including 20×32-bit instruction register file, a 32×8-bit regular register file and 32×16-bit constant register file, as shown in Table VI. For area comparison, the CPU includes 32kB of data memory, 4kB of instruction memory, and 1 kB of instruction cache, which is equivalent to the design parameters of the IPA. Table V presents the code-size (instructions and constants) of all the kernels used in the following experiments. The cost of the IRF is considered both in size and power. Thanks to the simpler architecture and tiny processing elements, at the target operating voltage of 0.6V, the IPA runs at 100 MHz while or1k can only reach 45MHz in the same operating point.

Figure 9 shows the area of the whole array and memory with different numbers of TCDM banks, where the total amount of memory is kept constant at 32kB. As the area of LSUs is negligible if compared to the overall system area, we show the area results for the worst-case scenario with maximum number of LSUs present in the PE array (i.e. 16). As shown in Figure 9, in the minimal configuration with 4 TCDM banks, the IPA area is dominated by that of the array (60%) and by the local data storage (35%), while the remaining 5% is consumed by the interconnect. Increasing the number of TCDM banks imposes a significant area overhead on the size of the interconnect. Also, the area of the TCDM increases as well due to the higher area/bit of small SRAM cuts necessary to implement 32kB of memory with several banks. Hence, it is fundamental to properly balance the number of LSUs and TCDM banks with the bandwidth requirements of applications.

B. Comparison of the proposed compilation approach with state of the art predication techniques

To evaluate the efficiency of the register allocation approach to handle the control flow we compare the execution of six control intensive kernels compared to the state of the art partial and full predication techniques. The results, presented in Table VII, show that the register based approach achieves a maximum of 1.33× (with minimum of 1.04× and average of 1.13×) and 1.8× (with minimum of 1.37× and average of 1.59×) performance gain compared to partial predication and full predication techniques. The maximum gain achieved over existing methods are highlighted in bold in the table. The smaller number of executed instructions allows the register allocation approach to outperform the partial and full predication

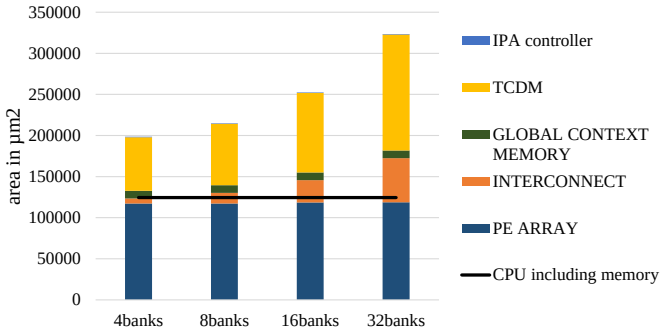


Figure 9: Synthesized area of IPA for different number of TCDM banks

techniques by an average of 1.54× (with min 1.35×, max 2×) and 1.71× (with min 1.44×, max 2×) respectively in terms of energy efficiency. The table also presents a comparison with respect to or1k CPU and C64 DSP processor [22] from TI. The register allocation approach achieves a maximum of 3.94×, 15.8× performance gain and 7.52×, 32.77× energy gain over or1k and C64 processor, respectively. Due to the abundance of branches in these kernels, the DSP processor performs worst. Finally, we compare with the basic systematic load-store (SLS) based approach for control mapping. It is depicted from the Table VII that the register allocation approach performs an average of 1.16× (with max of 1.46×, min of 1.05×) better than the SLS based approach, while gaining an average of 1.31× energy efficiency with a maximum gain of 2× and minimum gain of 1.07×.

C. Architectural Exploration

This section provides an extensive comparison with respect to the CPU computational model and an evaluation of the performance of the IPA while varying the number of LSUs and TCDM banks, a critical parameter for data-hungry accelerators. To carry out the exploration, we selected 7 compute intensive signal processing kernels featuring a high bandwidth towards the TCDM.

1) Performance: Generally speaking, the IPA performs well when significant parallelism can be extracted from a kernel. This concept is well shown in Figure 10, which compares the performance of the IPA with that of the or1k processor on a matrix multiplication when growing the size of the matrices from 2×2 to 32×32. It is possible to note that the increase of the kernel size increases the average utilization of the PEs as well, which in turn helps to enhance performance. It also demonstrates that the initial configuration time, which is dominant for small kernel size is well amortized for larger kernels, further contributing to improve performance.

Figure 11 presents the total execution time (clock cycles) of seven compute-intensive kernels. The execution time is

Table VII: Performance comparison between the register allocation approach and the state of the art approaches

Kernels	# loops	# condi tionals	Performance (cycles)						Energy(μJ)					
			CGRA				CPU	C64 DSP	CGRA				CPU	C64 DSP
			reg based	SLS based	partial pred	full pred			reg based	SLS based	partial pred	full pred		
cordic	1	2	328	408	396	542	513	286	0.001	**0.002**	**0.002**	**0.002**	0.004	0.002
sobel	4	11	179617	**262282**	188253	245583	454028	669794	0.736	1.102	1	1.058	3.531	5.656
gcd	1	1	55312	58596	**73747**	92852	67545	92184	0.227	0.246	0.392	0.4	0.525	0.778
sad	2	1	15962	16824	16573	**28776**	**62932**	**252193**	0.065	0.071	0.088	0.124	**0.489**	**2.13**
deblocking	5	7	472258	495081	518722	727243	834683	1310220	1.936	2.079	2.754	3.134	6.492	11.064
manh-dist	1	1	6288	6826	6738	9522	15394	55317	0.026	0.029	0.036	0.041	0.12	0.467
	max gain			1.46×	1.33×	1.8×	3.94×	15.8×		2×	2×	2×	7.52×	32.77×

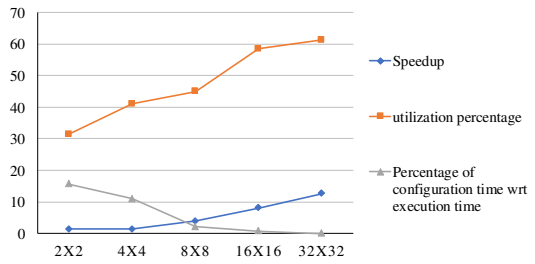


Figure 10: Performance of IPA executing matrix multiplication of different size

normalized with respect to that of or1k processor, where the kernels are compiled with -O3 optimization flag. The IPA outperforms the CPU by up to 20.3×, with an average speed-up of 9.7×. A quantitative performance comparison with respect to the CPU is presented in Table VIII. The table presents the configuration and execution cycles in the IPA for different kernels. It also presents the average utilization of PEs over the total execution period and total number of instructions executed in the IPA. The instruction count includes the instructions that are replicated on all the active PEs for keeping the PE in synch across conditionals and jumps. It also includes NOPs that are used when some PEs are stalled due to manipulation of index variables. However, during NOP execution PEs are clock gated and do not consume dynamic power. The IPA achieves a maximum of 18× and an average of 9.23× energy gain over the CPU.

To establish the impact of the memory bandwidth over performance and energy efficiency, we vary the number of LSUs in the PE array from 4 to 16 and the number of TCDM banks from 4 to 32. The number of LSUs defines the available bandwidth from the TCDM to the array, while increasing the number of TCDM banks reduces the banking conflict probability, improving performance. To perform the exploration without any bias towards configurations, the innermost loops of the kernels are unrolled to get a maximum of 16 load-store operations in one cycle (as the highest number of LSUs considered is 16, in the exploration). In Figure 11, each configuration is represented as a 2-dimensional number, where the first one represents the number of LSUs, and the second one represents the number of TCDM banks.

Results show that, as opposed to tightly coupled clusters of processors which require a banking factor of 2 (i.e. number of TCDM banks is twice the number of cores) [47], IPA performance is almost insensitive to the number of TCDM banks, and a configuration with a banking factor of 0.5 is sufficient to minimize the impact of contention on the shared memory banks for most applications. Indeed, while the typical processor execution requires several load/store operations for variables exceeding the size of the register file, direct CDFG mapping on the IPA does not add extra memory operations except primary inputs and outputs, since all the temporary variables are stored in the register file of the PEs. Moreover, flexible point-to-point connections within the array allow to efficiently exchange data among PEs, further reducing the pressure on the TCDM. This concept is well explained in Figure 4 and Figure 1, which show the typical mapping of an application on the IPA.

2) Energy Efficiency: Figure 12 shows the average breakdown of power consumption for different configurations of the IPA. As expected, the PE array is the most dominant power consumer for all the configurations. The configurations with 4 TCDM banks achieve the best power advantages in each group, since increasing the number of TCDM banks increases the complexity of the interconnect, causing timing pressure on the array, which increases the sizing of the cells, hence power consumption.

Figure 13 shows the average energy efficiency (MOPS/mW) for different configurations. Million Operations Per Second (MOPS) only considers the active PEs during execution, since a PE may be idle due to TCDM bank access conflicts, consecutive NOPs, or not mapped (not used in the application execution). Executions with high number of active PEs/cycle achieve large MOPS. As depicted in Figure 13, for different number of LSUs in the PE array, the configuration with 4 TCDM banks achieves the best energy efficiency, since this is the least number of banks in each configuration, it causes lowest power consumption. At the same time, the active number of PEs/cycle does not get significantly impacted due to the least memory access policy of the compilation. As a result, the best efficiency is achieved at 2306 MOPS/mW for matrix multiplication, in a configuration with 8 LSUs and 4 TCDM banks. The minimum energy efficiency is achieved at 1112 MOPS/mW for separable filter in a configuration with 4 LSUs and 16 TCDM banks.

To investigate the power gain in the fine-grained clock gating we present the energy consumption of the clock gated IPA and the non clock gated IPA in Table VIII. In an average, the clock gated design consumes an average of 2× less power compared to that of the non clock gated design. Due to the regular architecture of the PE array, fine grained power management is much more suitable to implement than a processor. Moreover, thanks to the efficient execution of CDFG on the

Table VIII: Overall instructions executed and energy consumption in IPA vs CPU

Kernels		FIR	MatM (16×16)	Convolution	SepFilter	NonSepFilter	FFT	DC Filter
IPA	Configuration cycles	71	88	88	90	98	87	145
	Execution cycles	6071	11940	56241	827685	1852382	8076	4748
	Total number of instructions executed	44294	110946	531815	7349843	17486486	76310	28868
	Active PEs/cycle (%)	46.1	58.5	59.2	55.5	59	59.7	39.5
	Energy (μJ)	0.022	0.043	0.202	2.98	6.669	0.032	0.017
	Energy (μJ) in non-clock-gated IPA	0.047	0.077	0.479	7.152	11.704	0.063	0.045
CPU	Execution cycles	37677	96256	616805	5982730	9084101	164480	50085
	Energy (μJ)	0.132	0.337	2.159	20.94	31.794	0.576	0.175
	Speed-up	6.21x	8.06x	10.97x	7.23x	4.9x	20.3x	10.55x
	Energy-gain	6x	7.84x	10.69x	7.03x	4.77x	18x	10.29x

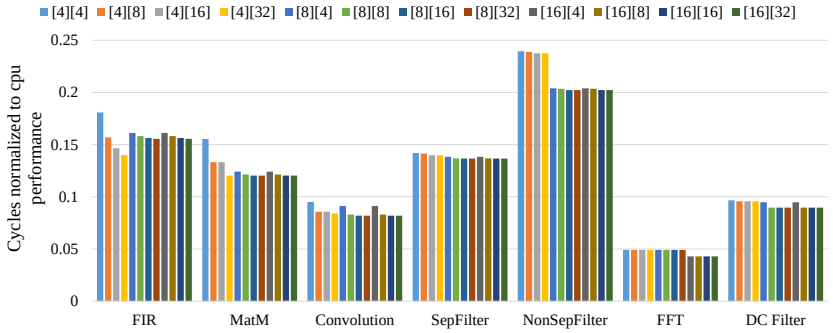


Figure 11: Latency performance in different configurations ([#LSUs][#TCDM Banks])

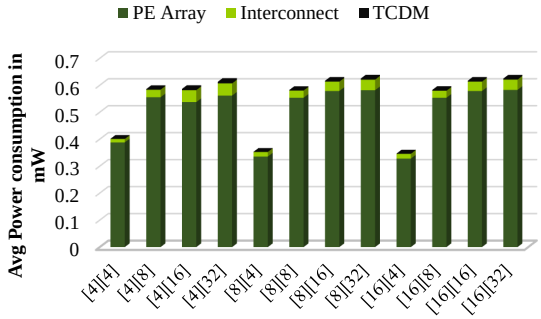


Figure 12: Average power breakdown in different configurations ([#LSUs][#TCDM Banks])

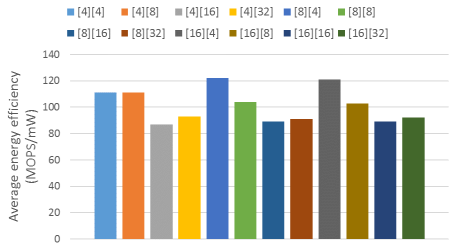


Figure 13: Average energy efficiency for different configurations ([#LSUs][#TCDM Banks])

array, the smaller energy required to execute an instruction in the IPA with respect to a CPU (5.6E-07 μJ vs 3.49E-06 μJ), and the effectiveness of the fine-grained power management the IPA outperforms the or1k CPU's energy efficiency by up to 18× (Table VIII). The energy per instruction execution in the IPA is much less than that of the CPU due to its simple instruction set architecture. Also, the lower number of memory operations executed in the IPA helps reducing on the average energy consumption.

D. Comparison with low-power CGRA architectures

Table IX shows a comparison with ~~low-power~~ existing CGRAs. For some papers, energy efficiency figures could not be extracted, so 'NA' is put in the corresponding cell. The energy efficiency figures of the other architectures are provided both in the original manufacturing technology node and scaled to the 28nm technology, according to the power scaling factor $C * V^2$. C and V represent the effective capacitance (approximated with the channel length of the technology) and the supply voltage of the designs, normalized to the nominal parameters of the 28nm technology node. It should be noted that this simplified scaling factor penalizes our design, since deep-submicron technologies such as 28nm, where the load capacitance of gates is typically dominated by wires require much more buffering than mature technology nodes, which penalizes energy efficiency. Nevertheless, IPA provides leading-edge energy efficiency, surpassing by more than one order of magnitude other architectures (ADRES, Morphosys, XPP, AsAP) featuring a C based mapping flow. The driving factors for this gain are (a) architectural simplicity with less complex interconnect network, (b) low power instruction processing, (c) lowest possible number of memory operations in application execution, (d) fine grained power management architecture, described in previous sections. Compared to ultra-low power targets (that fit in a power envelope of 3mW), the IPA presents a much better energy efficiency over [33] and [36] for which information could be extracted from the papers. One

[3]PEs perform 8-bit operations, hence energy efficiency is normalized to equivalent 32-bit operations, does not include the power of controlling processor.

Table IX: Comparison with the state of the art low power targets

Ref	Arch	Maps	Source	Access local memory	Tech [nm]	Supply voltage	Area [mm^2]	Power [mW]	Freq [MHz]	Area eff [MOPS /mm^2]	Energy eff [MOPS /mW]	Energy eff scaled to 28nm tech [MOPS/mW]	Perf [MOPS]
High performance targets													
[6]	Morphosys	DFG	ANSI C	PE	150	1.8V	256	4000	450	113	7.20	150	28800
[6]	Imagine	DFG	NA	PE	150	1.5V	144	4000	296	165	12.40	150	23700
[54]	RAW	CDFG	ANSI C	PE	150	1.8V	256	2288	100	NA	NA	NA	NA
[50]	TRIPS	DFG	NA	PE	130	1.0V	336	35868	366	NA	NA	NA	NA
[6]	ReMAP	CDFG	NA	PE	180	1.62V	8.28	312	200	386	10.30	173	3200
Low power targets													
[25]	TCPA	CDFG	Customized	VLIW	90	1.0V	15	12.48	200	106	112.00	360	1587
[46]	Layers	CDFG	NA	PE	65	1.0	0.35	44.45	488	2786	21.94	72	975
[29]	SmartCell	CDFG	Customized	PE	130	1.0V	8.2	160	100	13.04	37.8	176	6048
[17]	PipeRench	DFG	Customized	PE	180	1.8V	55.5	675	120	NA	NA	NA	NA
[41]	SYSCORE	CDFG	NA	PE	90	1.0V	5.73	18.5	100	NA	NA	NA	NA
[2]	ADRES	DFG	ANSI C	VLIW	90	1.0V	15	80	100	94	17.51	56	1409
[3]	XPP	CDFG	ANSI C	PE	90	1.0V	42	93	150	310	10.00	32	13000
[56]	AsAP	CDFG	ANSI C	PE	180	1.8V	23.76	84	116	40	11.00	229	942
[49]	MUCCRA-3	DFG	Customized	VLIW	65	1.2V	8.82	11	41.4	NA	NA	NA	NA
Ultra-low power targets													
[33]	Lopes et al	DFG	NA	PE	90	1.0V	0.45	3.47	100	222	28.8	92.6	100
[36]	CMA	DFG	Customized	μC	65	0.5V	25	1.6	85	[3]3	[3]186	[3]430	[3]74
[14]	SIMD-CGRA	DFG	ANSI C	PE	65	0.9	0.59	NA	1	NA	NA	NA	NA
[23]	ULP-SRP	DFG	ANSI C	VLIW	40	0.5V	NA	0.21	7	NA	NA	NA	NA
This paper	IPA	CDFG	ANSI C	PE	28	0.6V	0.25	0.49	100	3036	1617	1617	759

distinguishing characteristic of the proposed accelerator is the flexible execution model capable of implementing CDFG on the array without the need of a host processor, coupled with a fully automated mapping flow that starts from a plain ANSI C description of the application. Moreover, the memory architecture, based on a shared multi-banked TCDM enables easy integration within ultra-low-power tightly coupled clusters of processors, while fine-grained power management allows improving energy efficiency by up to 2×. The average power consumption on the IPA is 0.49mW, which is compatible with the ultra-low power target.

VI. Conclusion

This work presents an ultra-low power coarse grained reconfigurable array accelerator for near-sensor processing. The proposed *Integrated Programmable-Array* (IPA) is a 2-D array of NxN processing elements (a 4x4 configuration is used in this paper), and leverages a multi-banked tightly coupled data memory for data storage, to ease the integration in clustered multi-core architectures. We present a compilation flow targeting the mapping of both control and data flow portions of kernels onto the array of processing elements, aimed at reducing the pressure on the shared data memory, along with an architectural exploration of the memory architecture parameters. The results of the exploration show that a configuration of the IPA with 8 load-store units and 4 TCDM banks achieves the optimal performance/energy trade-off featuring an average speed-up of 9.7× (max 20.3×, min 4.9×) compared to a general-purpose processor. With respect to state of the art partial and full predication techniques, the proposed compilation flow improves performance by 1.54× on average (min 1.35×, max 2×) and energy efficiency by 1.71× on average (min 1.44×, max 2×). Thanks to the optimized architecture and mapping flow, the proposed accelerator achieves an average energy efficiency of 1617 MOPS/mW over a wide range of sensor signal processing kernels, surpassing other CGRA architectures featuring a C based mapping flow by more than one order of magnitude.

Acknowledgements

This work was funded by the ERC MultiTherman Project (ERC-AdG-291125), and by the OPRECOMP Project funded from the European Unions Horizon 2020 Research and Innovation Programme under grant agreement No. 732631. We also thank STMicroelectronics for granting access to the FDSOI 28nm technology libraries.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[2] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the adres coarse-grained reconfigurable array. In *Proceedings of the 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] F. Campi, R. König, M. Dreschmann, M. Neukirchner, D. Picard, M. Jüttner, E. Schler, A. Deledda, D. Rossi, A. Pasini, M. Hübner, J. Becker, and R. Guerrieri. RTL-to-layout implementation of an embedded coarse grained architecture for dynamically reconfigurable computing in systems-on-chip. In *2009 International Symposium on System-on-Chip*, pages 110–113, Oct 2009.

[4] L. Chen and T. Mitra. Graph minor approach for application mapping on cgras. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):21, 2014.

[5] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 379–384. ACM, 2012.

[6] P. Dai, X. Wang, X. Zhang, Q. Zhao, Y. Zhou, and Y. Sun. A high power efficiency reconfigurable processor for multimedia processing. In *2009 IEEE 8th International Conference on ASIC*, pages 67–70, Oct 2009.

[7] S. Das, K. J. M. Martin, P. Coussy, D. Rossi, and L. Benini. Efficient mapping of CDFG onto coarse-grained reconfigurable array architectures. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132, Jan 2017.

[8] S. Das, T. Peyret, K. Martin, G. Corre, M. Thevenin, and P. Coussy. A scalable design approach to efficiently map applications on CGRAs. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 655–660, July 2016.

[9] S. Das, D. Rossi, K. Martin, P. Coussy, and L. Benini. A 142 mops/mw integrated programmable array accelerator for smart visual processing. In *2017 IEEE International Symposium of Circuits and Systems (ISCAS)*, page Accepted, 2017.

[10] B. De Sutter, P. Raghavan, and A. Lambrechts. Coarse-grained reconfigurable array architectures. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems*, pages 449–484. Springer US, 2010.

[11] M. Dehyadegari, A. Marongiu, M. R. Kakoee, S. Mohammadi, N. Yazdani, and L. Benini. Architecture support for tightly-coupled multi-core clusters with shared-memory hw accelerators. *IEEE Transactions on Computers*, 64(8):2132–2144, 2015.

[12] G. Donohoe. Reconfigurable data path processor, Apr. 19 2005. US Patent 6,883,084.

[13] G. W. Donohoe, D. M. Buehler, K. J. Hass, W. Walker, and P.-S. Yeh. Field programmable processor array: Reconfigurable computing for space. In *2007 IEEE Aerospace Conference*, pages 1–6. IEEE, 2007.

[14] L. Duch, S. Basu, R. Braojos, G. Ansaloni, L. Pozzi, and D. Atienza. Heal-wear: An ultra-low power heterogeneous system for bio-signal analysis. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 64(9):2448–2461, 2017.

[15] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini. Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–14, 2017.

[16] M. Gautschi, A. Traber, A. Pullini, L. Benini, M. Scandale, A. Di Federico, M. Beretta, and G. Agosta. Tailoring instruction-set extensions for an ultra-low power tightly-coupled cluster of openrisc cores. In *2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 25–30, Oct 2015.

[17] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[18] M. Hamzeh, A. Shrivastava, and S. Vrudhula. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures (cgras). In *Proceedings of the 50th Annual Design Automation Conference*, page 18. ACM, 2013.

[19] K. Han, J. Ahn, and K. Choi. Power-efficient predication techniques for acceleration of control flow execution on cgra. *ACM Trans. Archit. Code Optim.*, 10(2):8:1–8:25, May 2013.

[20] K. Han, J. K. Paek, and K. Choi. Acceleration of control flow on cgra using advanced predicated execution. In *Field-Programmable Technology (FPT), 2010 International Conference on*, pages 429–432, Dec 2010.

[21] K. Han, S. Park, and K. Choi. State-based full predication for low power coarse-grained reconfigurable architecture. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1367–1372, March 2012.

[22] T. Instruments. Tms320c64x/c64x+ dsp cpu and instruction set reference guide. *Texas Instruments, User manual SPRU732C*, 2005.

[23] C. Kim, M. Chung, Y. Cho, M. Konijnenburg, S. Ryu, and J. Kim. Ulp-srp: Ultra low-power samsung reconfigurable processor for biomedical applications. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(3):22, 2014.

[24] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi. Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization. In *Design, Automation and Test in Europe*, pages 12–17. IEEE, 2005.

[25] D. Kissler, A. Strawetz, F. Hannig, and J. Teich. Power-efficient reconfiguration control in coarse-grained dynamically reconfigurable architectures. *Journal of Low Power Electronics*, 5(1):96–105, 2009.

[26] D. Lampret, C.-M. Chen, M. Mlinar, J. Rydberg, M. Ziv-Av, C. Ziomkowski, G. McGary, B. Gardner, R. Mathur, and M. Bolado. Openrisc 1000 architecture manual. *Description of assembler mnemonics and other for OR1200*, 2003.

[27] J. Lee, S. Seo, H. Lee, and H. U. Sim. Flattening-based mapping of imperfect loop nests for cgras? In *2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Oct 2014.

[28] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo*, 9(4):341–352, 1973.

[29] C. Liang and X. Huang. Smartcell: An energy efficient coarse-grained reconfigurable architecture for stream-based applications. *EURASIP Journal on Embedded Systems*, 2009(1):518659, 2009.

[30] C. Liu, H. Ng, and H. K. So. Automatic nested loop acceleration on fpgas using soft CGRA overlay. *CoRR*, abs/1509.00042, 2015.

[31] D. Liu, S. Yin, L. Liu, and S. Wei. Polyhedral model based mapping optimization of loop nests for cgras. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–8. IEEE, 2013.

[32] L. Liu, J. Wang, J. Zhu, C. Deng, S. Yin, and S. Wei. Tlia: Efficient reconfigurable architecture for control-intensive kernels with triggered-long-instructions.

[33] J. Lopes, D. Sousa, and J. C. Ferreira. Evaluation of cgra architecture for real-time processing of biological signals on wearable devices. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–7, Dec 2017.

[34] K. T. Madhu, S. Das, N. S., S. K. Nandy, and R. Narayan. Compiling HPC Kernels for the REDEFINE CGRA. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 405–410, Aug 2015.

[35] A. Marshall, T. Stansfield, I. Kostarnov, J. Vuillemin, and B. Hutchings. A reconfigurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 135–143. ACM, 1999.

[36] K. Masuyama, Y. Fujita, H. Okuhara, and H. Amano. A 297mops/0.4 mw ultra low power coarse-grained reconfigurable accelerator CMA-SOTB-2. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[37] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. In *Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on*, pages 166–173. IEEE, 2002.

[38] E. Mirsky, A. DeHon, et al. Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources. In *FCCM*, volume 96, pages 17–19, 1996.

[39] N. Ozaki, Y. Yoshihiro, Y. Saito, D. Ikebuchi, M. Kimura, H. Amano, H. Nakamura, K. Usami, M. Namiki, and M. Kondo. Cool mega-array: A highly energy efficient reconfigurable accelerator. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–8, Dec 2011.

[40] H. Park, K. Fan, S. A. Mahlke, T. Oh, H. Kim, and H.-s. Kim. Edge-centric modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 166–176. ACM, 2008.

[41] K. Patel, S. McGettrick, and C. J. Bleakley. Syscore: A coarse grained reconfigurable array architecture for low energy biosignal processing. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual International Symposium on*, pages 109–112. IEEE, 2011.

[42] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[43] T. Peyret, G. Corre, M. Thevenin, K. Martin, and P. Coussy. Efficient application mapping on cgras based on backward simultaneous scheduling/binding and dynamic graph transformations. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 169–172. IEEE, 2014.

[44] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini. A fully-synthesizable single-cycle interconnection network for shared-l1 processor clusters. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[45] Z. E. Rakossy, A. Acosta-Aponte, T. G. Noll, G. Ascheid, R. Leupers, and A. Chattopadhyay. Design and synthesis of reconfigurable control-flow structures for cgra. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2015.

[46] Z. E. Rákossy, D. Stengele, G. Ascheid, R. Leupers, and A. Chattopadhyay. Exploiting scalable cgra mapping of lu for energy efficiency using the layers architecture. In *Very Large Scale Integration (VLSI-SoC), 2015 IFIP/IEEE International Conference on*, pages 337–342. IEEE, 2015.

[47] D. Rossi, A. Pullini, I. Loi, M. Gautschi, F. K. Gürkaynak, A. Bartolini, P. Flatresse, and L. Benini. A 60 GOPS/W, -1.8 V to 0.9 V body bias

ULP cluster in 28 nm UTBB fd-soi technology. *Solid-State Electronics*, 117:170 – 184, 2016.

[48] Y. Saito, T. Sano, M. Kato, V. Tunbunheng, Y. Yasuda, M. Kimura, and H. Amano. Muccra-3: a low power dynamically reconfigurable processor array. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 377–378. IEEE Press, 2010.

[49] Y. Saito, T. Sano, M. Kato, V. Tunbunheng, Y. Yasuda, M. Kimura, and H. Amano. Muccra-3: a low power dynamically reconfigurable processor array. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 377–378. IEEE Press, 2010.

[50] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ilp, tlp, and dlp with the polymorphous trips architecture. In *ACM SIGARCH Computer Architecture News*, volume 31, pages 422–433. ACM, 2003.

[51] H. Singh, M.-H. Lee, G. Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, May 2000.

[52] Y. Song and Y. Lin. Unroll-and-jam for imperfectly-nested loops in dsp applications. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 148–156, New York, NY, USA, 2000. ACM.

[53] H. Su, Y. Fujita, and H. Amano. Body bias control for a coarse grained reconfigurable accelerator implemented with silicon on thin box technology. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, Sept 2014.

[54] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal. A 16-issue multiple-program-counter microprocessor with point-to-point scalar operand network. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 170–171 vol.1, Feb 2003.

[55] S. Yin, P. Zhou, L. Liu, and S. Wei. Acceleration of nested conditionals on cgras via trigger scheme. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 597–604. IEEE Press, 2015.

[56] Z. Yu, M. J. Meeuwsen, R. W. Apperson, O. Sattari, M. Lai, J. W. Webb, E. W. Work, D. Truong, T. Mohsenin, and B. M. Baas. AsAP: An asynchronous array of simple processors. *IEEE Journal of Solid-State Circuits*, 43(3):695–705, 2008.