SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

January 2025, Vol. 68, Iss. 1, 112402:1–112402:13
https://doi.org/10.1007/s11432-023-3965-1

# Software-defined process-near-memory architecture using 3D hybrid bonding integration

Anlin XU[1,2], Chenchen DENG[3*], Jianfeng ZHU[1], Yao WANG[1],
Shaojun WEI[1,3] & Leibo LIU[1,3]

[1]*School of Integrated Circuits, Tsinghua University, Beijing 100084, China;*
[2]*Beijing Institution of Tracking and Communication Technology, Beijing 100094, China;*
[3]*Beijing National Research Center for Information Science and Technology, Tsinghua University, Beijing 100084, China*

Received 2 September 2023/Revised 23 November 2023/Accepted 27 February 2024/Published online 17 December 2024

**Abstract** With the unprecedented explosive growing amount of global data, the development of computing chips, which encounter bottlenecks such as power wall and memory wall, cannot satisfy the demanding requirement. This work proposes a software-defined process-near-memory (SDPNM) computing architecture implemented using 3D hybrid bonding integration. The software-defined chip architecture, featuring spatial computations and dynamic reconfiguration, innovates in a top-down manner to achieve high energy efficiency while maintaining flexibility after fabrication. The process-near-memory integration further advances the SDNPM chip in a bottom-up way to reduce the energy consumption of data movement while improving the bandwidth. Utilizing a relatively mature fabrication and bonding process can result in feasible solutions for both data-intensive and compute-intensive applications including digital signal processing and artificial intelligence. The logic die is fabricated in the SMIC 40 nm process and the DRAM die is fabricated in the PSMC 25 nm process. The hybrid bonding is implemented by XMC. The experimental results show that the energy efficiency of the proposed SDPNM chip is 33.1× better than the state-of-the-art FPGA ranging from 8.2× to 104.1×.

**Keywords** software-defined chips, process near memory, energy efficiency, dynamic reconfiguration, domain specific

**Citation** Xu A L, Deng C C, Zhu J F, et al. Software-defined process-near-memory architecture using 3D hybrid bonding integration. Sci China Inf Sci, 2025, 68(1): 112402, https://doi.org/10.1007/s11432-023-3965-1

## 1 Introduction

With the rapid development of emerging applications such as artificial intelligence (AI), and the internet of things, the amount of data to be processed grows exponentially, which is a stringent demand for computing chips to improve the performance dramatically [1]. For the past few years, the improvement in the performance of computing chips mainly relied on the advancement of the integrated circuit fabrication process. However, with the process entering sub 2 nm technology, Moore's law and Dennard scaling law nearly come to an end [2]. It is rather difficult to continuously benefit from the fabrication technology, and the power wall has become a major constraint of computing chips. Much attention has been paid to the energy efficiency along with the performance [3]. In the meanwhile, the flexibility of the computing architecture becomes an important metric. The rapid emergence of new applications and fast upgrading speed of software makes the hardware implementation that cannot adapt to software changes face major challenges such as short life cycles and unbearable escalating non-recurring engineering costs. Therefore, the energy efficiency and flexibility of computing architectures are becoming essential design considerations.

To satisfy all the metrics simultaneously, mainstream computing architectures are facing severe challenges. Figure 1 compares the characteristics of the four types of computing architectures in terms of programmability in both temporal and spatial domains. Application-specific integrated circuits (ASICs) have extreme energy efficiency but lack flexibility. Instruction-based computing chips such as general-purpose processors (GPPs), and graphics processing units (GPU) are software programming. Software implementations are sequential and allow temporal switching with high flexibility. Application algorithms

* Corresponding author (email: chenchendeng@tsinghua.edu.cn)

© Science China Press 2024

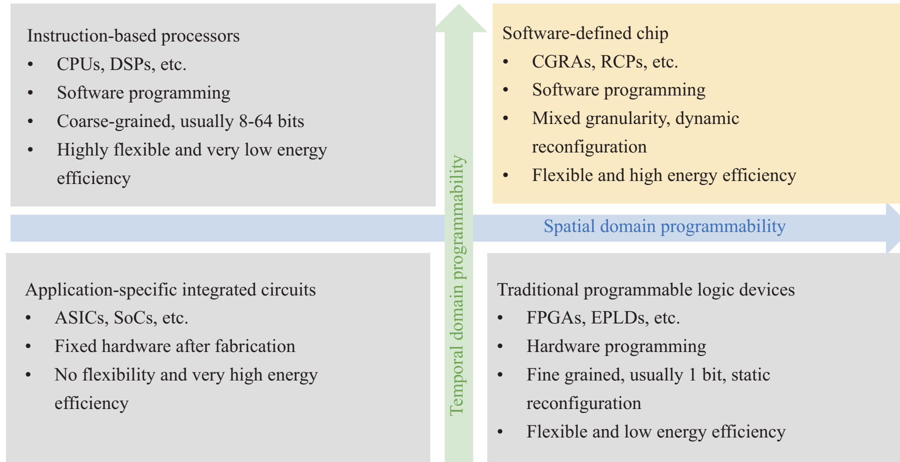


**Figure 1** (Color online) Comparison of different types of computing architectures.

are converted to instructions without fully exploiting the advantages of the hardware. Therefore, they are of great flexibility but with low energy efficiency. Traditional programmable logic devices represented by field programmable gate arrays (FPGAs) are hardware programming. Hardware resources are abstracted to several spatially distributed functional clusters and the combination of these clusters can realize different applications. However, the single-bit programming granularity and static configuration result in several problems such as low energy efficiency, limited capacity, and high usage threshold. Software-defined chips (SDCs) are a novel paradigm involving both hardware and software of computing chips. By bridging the gap between the software and the hardware, the function of the hardware can be directly defined by the software in the runtime. The computing architecture can be configured into different functions both in the temporal and spatial domain. The optimizations can also be realized in real-time, and in this way, the high energy efficiency of the hardware and the high flexibility of the software are combined into a whole, which makes SDCs a promising candidate to satisfy the above mentioned key metrics [4, 5]. Much research has been carried out in SDCs featured with coarse-grained reconfigurable architecture [6–8], but the studies are mainly focused on the reconfigurable processing units.

In addition to the energy efficiency of data processing, the energy consumption of memory access is also crucial. Modern computing systems are mainly based on traditional von Neumann architecture, the data to be processed are moved between the processing unit and memory, which results in high overheads in energy consumption. For a processor in 40 nm technology, the energy consumption of computation only takes less than 10% of the overall consumption while the rest 90% is for memory access and control [9]. One solution is to constantly improve the memory bandwidth and data rate of off-chip memory, such as the high-bandwidth memory (HBM) dynamic random access memory (DRAM) [10]. Featured with 3-D stacked multiple DRAMs and base logic die, HBM increases the number of connections between logic and memory with interposer and microbumps and has evolved for several generations into HBM3 which still incurs the significant overhead of the data movement [11]. To eliminate the costs of data movement, computing in-memory (CIM) has been proposed for various data-centric applications such as machine learning and scientific computing [12, 13]. For CIM, the energy efficiency for certain computation tasks is improved by the customized memory device to accommodate the specific computation pattern. This usually involves a dedicated device fabrication process, which makes mass production quite challenging. In addition, the flexibility to implement various tasks is also rather limited. A practical alternative solution is the process-near-memory architecture using commercially mature 3D hybrid bonding (HB) integration [14, 15]. The logic and memory dies are fabricated separately using the corresponding process and the logic-to-DRAM HB technology leads to high energy efficiency [16].

Before the concepts of near-memory and in-memory processing, SDCs usually used on-chip memory or multilayer cache for the processing array's data caching. This method uses a unified memory model and management mechanism, and abstracts the hardware architecture at a high level, with a concise and easy-to-use programming interface. However, this method brings the following two potential problems. First, the memory system is fixed and cannot perform dedicated optimization on the data structure of different applications, so it is only suitable for regular data access patterns and organization structures. The efficiency of irregular data structures may be very low. Second, the centralized or multilayer mem-

ory structure does not match the spatial computing characteristics of dynamically reconfigurable chips, resulting in low data transmission efficiency. A dynamically reconfigurable chip usually contains a series of reconfigurable processing units regularly distributed in the spatial domain, and some or even all of which can perform memory access. These memory accesses are often fragmented, and there may be a large number of redundant address calculations and other operations, which leads to a significant negative impact on performance and memory access efficiency.

This paper proposes a software-defined process-near-memory architecture (SDPNM) using 3D hybrid bonding integration, aiming for high-performance and high-efficiency computing while maintaining flexibility for domain-specific applications, especially for data-intensive and compute-intensive tasks. The main contribution of the paper can be summarized as follows. First, the energy efficiency of flexible computing chips is further improved by introducing the software-defined memory concept into the SDCs paradigm combined with commercially practical process-near-memory technology. To our best knowledge, this is the first silicon implementation of the software-defined process near memory architecture. Second, a compilation system for agile and efficient mapping various tasks onto the hardware architecture is designed, and users without knowing the details of the underlying hardware structure can easily program the computing chip with a high-level language. Third, both the compute-intensive (such as digital signal processing) and data-intensive (such as AI models) are supported by the proposed computing architecture, and the experimental results show that SDPNM outperforms the state-of-the-art FPGA by 33.1× in average.

The rest of this paper is organized as follows. Section 2 focuses on the hardware part which involves the overall architecture of the proposed SDPNM and the optimization techniques. The software part including the compilation system and dynamical configuration methods are presented in Section 3. Section 4 presents the details of the fabricated chip and specifications as well as the experimental results and comparisons. Finally, the conclusion is made in Section 5.

## 2 System architecture

Figure 2 shows the overall architecture of the 3D stacked chip. The DRAM die is connected to the logic die via high-density hybrid bonding. The two dies are of the same dimension. The DRAM die consists of 12 identical 1 Gb units and each unit is composed of 8 independent 128 Mb blocks. Four 1 Gb units are disabled for the wiring of the logic die, and therefore the total capacity of the DRAM die is 1 GB. The memory controllers and PHYs of DRAM units are located on the logic die. In addition, there are five heterogeneous processing element arrays (PEAs) of three types, namely NPEA, FPEA, and SPEA, respectively.

The detailed architecture of SDPNM is shown in Figure 3. Five PEAs of three types are the computing engine of SDPNM and the main difference among the three types of PEAs is the operations that PEs can support and the DRAM access patterns. The main consideration of heterogeneous design is to improve energy and area efficiency based on the characteristics of the algorithms. NPEA is designed for adaptive filtering in digital signal processing algorithms such as cascaded integrator comb (CIC), and numerically controlled oscillator (NCO). PEs in NPEA support eight different operations such as add, and subtract, which are extracted from CIC and NCO, as listed in Table 1. Both FPEA and SPEA, with 27 different opcodes, can support commonly used algorithms in signal processing and deep learning, including correlation, fast Fourier transform (FFT), filtering and convolution, and matrix operation. These algorithms involve a large number of multiplications while CIC and NCO require no more than two multipliers. Therefore, the major difference is that each PE in FPEA and SPEA has four 16-bit multipliers while PEs in NPEA do not. This could save a lot of area and energy. With the DRAM units connected to SPEA via memory controllers, all 64 PEs in SPEA have direct access to the DRAM without conflicts while PEs in NPEA and FPEA can only access the DRAM via host controller or SPEA if necessary. Figure 4 shows the interconnections between five heterogeneous PEAs, and the PEAs are connected along the boundaries. The reconfigurable datapath design is tailored for different algorithm requirements. Take digital signal processing for example, the complete digital down converter (DDC) includes NCO, CIC and multiple stages of FIRs which could be mapped onto different PEAs respectively. However, sometimes NCO, CIC, or FIR are bypassed in certain application scenarios, and in such cases, the PEAs could be quickly configured into the desired datapath.

The internal structure of PEAs and PEs are shown in Figure 5. There are 64 PEs in the form of an 8×8

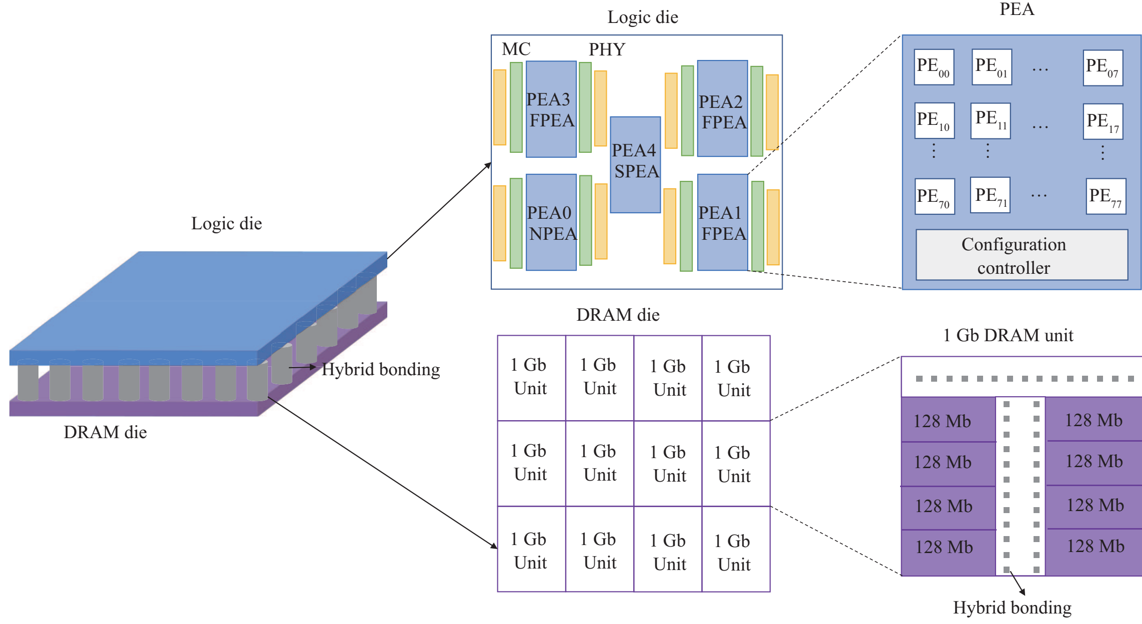


**Figure 2** (Color online) Overview of 3D-stacked chip and the layout diagram of the logic die and DRAM die.

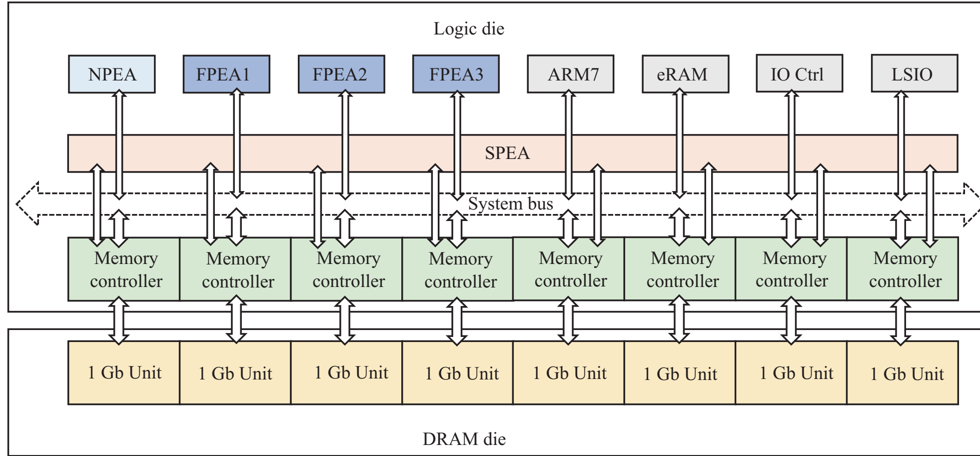


**Figure 3** (Color online) Structural diagram of SDPNM architecture.

**Table 1** Operation codes for NPEA, FPEA, and SPEA.

| NPEA | | FPEA/SPEA | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | in1+in2 | 1 | Add: in1+in2 | 10 | Bitrev: bitrev | 19 | CFa: complex coef fir |
| 2 | in1-in1_dly | 2 | Sub: in1−in2 | 11 | Cat: spl_s | 20 | RFb: real coef fir symmetry point |
| 3 | in1_l*sin(in1_h) | 3 | And: in1 & in2 | 12 | RSftS: r_sft_s | 21 | CFb: complex coef fir symmetry point |
| 4 | in1_l*cos(in1_h) | 4 | Or : in1 \| in2 | 13 | LSftS: l_sft_s | 22 | CMult: complex mult/complex mult add |
| 5 | (in1<<c_gain)-in1_dly(scale) | 5 | Xor: in1 ^in2 | 14 | AddSpl: add_spl_s | 23 | MCC: mult_cpl_conj |
| 6 | (in1-in1_dly)<<f_gain | 6 | Mux: in3 ? in1 : in2 | 15 | SubSpl: sub_spl_s | 24 | Cmac: mult_cpl_s_add |
| 7 | (in1+in1_dly)<<f_gain | 7 | Lls: in1<<in2 | 16 | Square: square | 25 | Cmsub: mult_cpl_s_sub |
| 8 | in1<<c_gain+in2 | 8 | Lrs: in1>>in2 | 17 | MultS: mult_s | 26 | MCCA: mult_cpl_conj_add |
| | | 9 | Cmp: min/max | 18 | RFa: real coef fir | 27 | MCCS: mult_cpl_conj_sub |

array, and both 1D and 2D interconnection among PEs are supported. The data and contexts are loaded externally, and the PEA parses the configuration memory and executes according to the configuration contexts. The PEA controller is in charge of the status of all PEs. Each PE consists of an arithmetic logic unit (ALU) and load store unit (LSU), the basic computing and storage units respectively. Each ALU has two 32-bit inputs and one 1-bit input. The two 32-bit inputs are operands of the coarse-grained

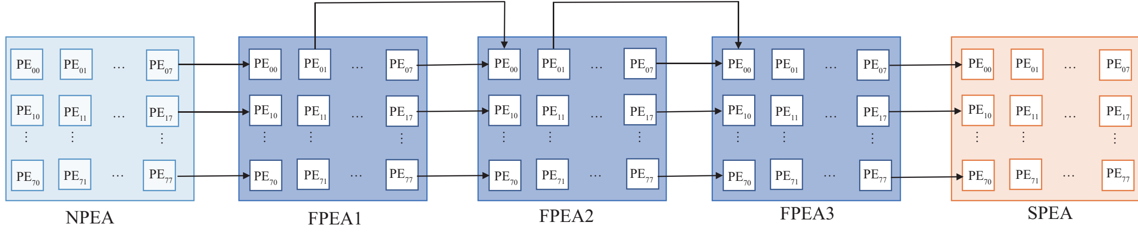


**Figure 4** (Color online) Interconnection between five heterogenous PEAs.

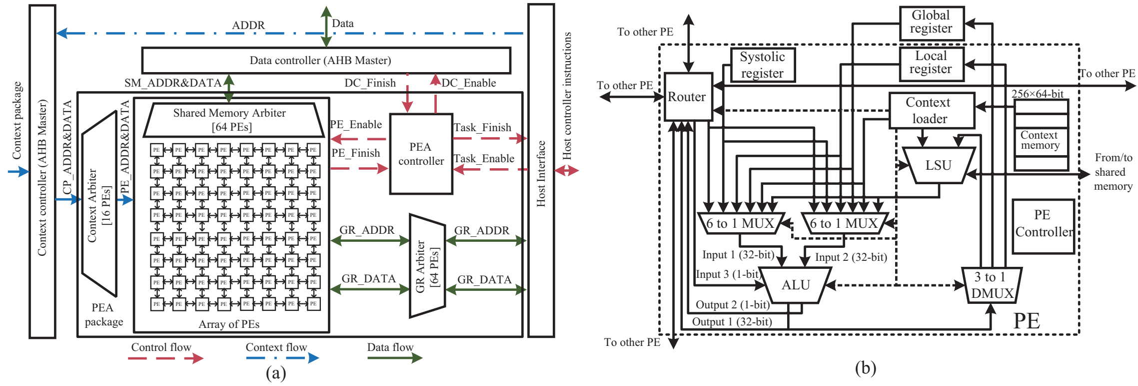


**Figure 5** (Color online) Architectures of PEA (a) and PE (b).

computation while the 1-bit input is for condition and carry-in. The source of the 32-bit inputs could be from the shared memory (input of the PEA computation), global register files (results from the host controller), the register files of this PE (intermediate results of this PE computation), output from this PE and other neighboring PEs (results from previous machine cycle), and immediate operands. The source of the 1-bit input is from other PE's 1-bit output. ALUs are designed to support multiple operations and are defined by a 5-bit operation code (Opcode) as listed in Table 1. The hardware utilization is improved by reusing function units in the ALU. For example, multipliers are reused for real, complex, and complex conjugate multiplication as well as complex multiplication accumulation by adding a few extra logic. PEs are mesh-connected via routers, and PEs can access the computation results of the last machine cycle from four adjacent PEs, four one-hop PEs, four PEs on the upper and lower rows (left and right) and itself. This leads to a more energy-efficient mechanism of data exchange without involving memory access.

The operation codes of each PE and the interconnections among the PEs are all defined in the configuration contexts, which are generated by the compilation system from a high-level language. Upon receiving the enable signal, the PEs execute the operations accordingly just like the ASICs that are driven by dataflow. Figure 6 shows two mapping examples of PEAs. As shown in Figure 6(a), four PEAs are constructed into a 2×2 array through configuration to implement 16×16 complex matrix multiplication. The arrays are interconnected at the boundaries and data flows in from horizontal and vertical directions through the systolic array to reduce the energy overhead and avoid conflicts of memory access. Since the scale of PEs and matrix are equivalent, there is no iteration in the configuration of PEAs or PEs. The opcode of all PEs is defined to be signed complex multiplication and accumulation. After the execution of the current package of configuration context, the function of PEs can be changed accordingly to implement other algorithms. As shown in Figure 6(b), one PEA is configured to implement FFT of a 1024-point complex, which involves 5 stages of Radix-4 butterfly computations. A Radix-4 butterfly computation can be implemented by a group of 8 PEs. For example, the opcode of PE0, 1, 8 and 9 is Add&Sub, while that of PE16, 17, 24 and 25 is signed complex multiplication and accumulation. The PEA array with 64 PEs can support eight paralleled Radix-4 butterfly computations simultaneously. The data is shared through the routers of adjacent PEs or global registers files to make sure that no memory access conflicts for 1024-point FFT.

There are eight 128 M banks in each 1 Gb unit and there are eight 1 Gb units available on the DRAM die. These 64 128 M banks correspond to the 64 PEs in SPEA and different banks can be accessed by the PEs. For each task, the memory resource is dynamically allocated and the memory access bandwidth is

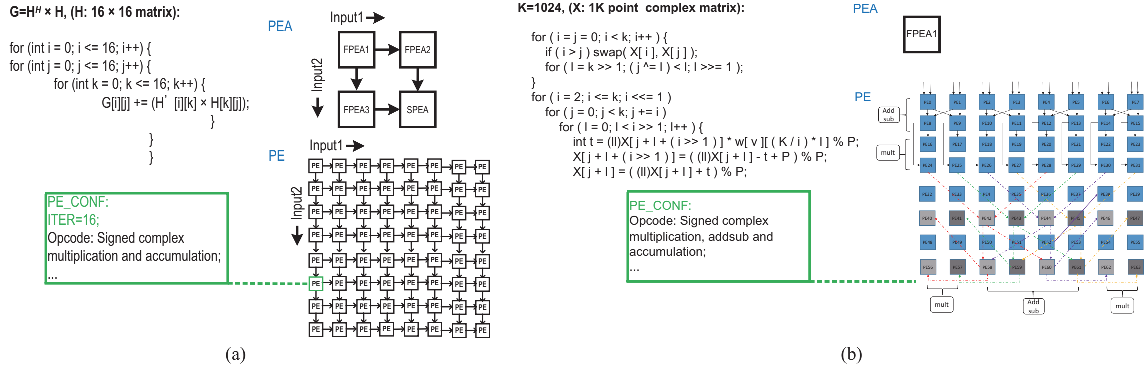


**Figure 6** (Color online) Mapping examples of 16 × 16 complex matrix multiplication (a) and FFT of 1024-pint complex (b).

utilized more efficiently. Also, the pipeline efficiency during data processing is improved by the memory interface optimization as the interruption of the pipeline caused by the memory access is greatly alleviated.

After the logic wafer and DRAM wafer are designed and fabricated, top vias and bottom vias are formed by a series of processes including planarization, photography, and dry etch. The logic wafer is then flipped over and bonded face-to-face to the DRAM wafer with hybrid bonding, which connects two wafers with Cu-Cu metal bonding at a temperature below 400°C. Hybrid bonding integration enables a short connection wire and much better density of vias compared with traditional integration methods such as microbumps. Then the Si substrate of the logic wafer is thinned to about 3 μm, and the PAD window is opened from the backside using TSV technology. In this way, both logic and DRAM wafers can be designed and optimized separately to achieve their best performance. Then high bandwidth with low energy consumption logic-to-memory interface is achieved via high density but low resistance hybrid bonding. More details can be further found in [14].

## 3 Compilation and configuration

As a novel chip architecture paradigm, SDCs are not just reconfigurable hardware but also involve software that can efficiently define the function of the hardware and dynamically optimize the performance. With the emerging applications' increasing demand for computing power, the scale of computing resources in the SDC grows exponentially. How to effectively utilize the hardware becomes essential for the SDPNM. The cost of manual mapping has seriously affected the development efficiency. Therefore, it is important to develop an automated compilation system for SDPNM. The compilation system translates the high-level language into the functionally equivalent binary. A desired compilation system can effectively tap the hardware potential of SDPNM without affecting the productivity of programmers, and provide an efficient channel to fully utilize the hardware resources. The overall compilation framework of SDPNM is shown in Figure 7, and it can briefly be divided into the compilation for PEAs and GPPs. The GPP compilation has been well developed and this paper will focus on the compilation of PEAs.

### 3.1 Task division

The PEAs, including a large number of PEs, are tailored for data-intensive and compute-intensive kernels but not for control-intensive kernels which interrupt pipelines in PEAs and reduce the overall efficiency. Therefore, the basic blocks that do not contain control flows are allocated on PEAs while the rest will be executed on GPPs. The machine codes, including configuration information of PEAs and assembly code of GPP, are obtained through the PEA compiler and the GPP compiler respectively, and then the machine code executable by the SDPNM is obtained through the assembler. However, too frequent data synchronization and communication between GPP and PEAs may seriously reduce the benefits of the division. Therefore, the premise that the above task division method is beneficial is that the communication overhead between GPP and PEAs is less than the performance improvement that PEAs can provide. For applications that are control-intensive or have limited parallelism, another way of dividing tasks across basic blocks can be used. Multiple basic blocks and control statements are combined into a hyperblock to be executed on PEAs. In this way, the parallelism is improved and the communication cost between GPP and PEA is reduced.

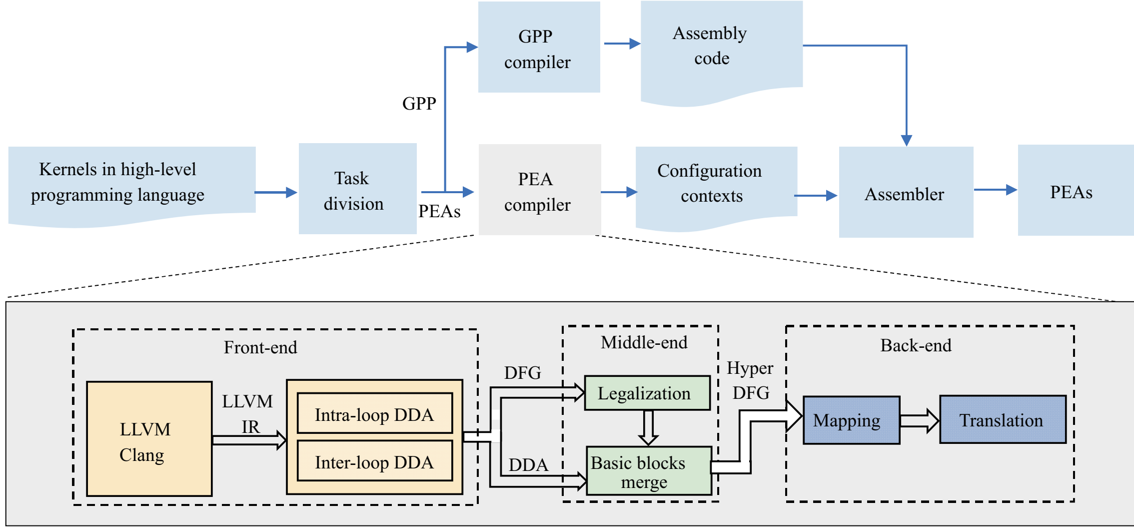


**Figure 7** (Color online) Compilation framework of the proposed SDPNM architecture.

**Algorithm 1** Example of inter-loop DDA

1: SM → L8.I8 load58 (27)
2: SM → L8.I8 load61 (28)
3: SM → L8.I8 load64 (29)
4: SM → L8.I8 load67 (30)
5: SM → L8.I8 load70 (31)
6: L7.I1 store4 → L8.I8 load73 (32)
7: L7.I2 store4 → L8.I8 load76 (33)
8: L7.I3 store4 → L8.I8 load79 (34)
9: L7.I4 store4 → L8.I8 load82 (35)
10: L7.I5 store4 → L8.I8 load85 (36)
11: L7.I6 store4 → L8.I8 load88 (37)
12: L7.I7 store4 → L8.I8 load91 (38)
13: L7.I8 store4 → L8.I8 load94 (39)

### 3.2 Front-end

After the task division, the kernels that are assigned to PEAs will be processed by the compiler front end. The compiler front-end is based on the LLVM compilation framework [17]. Preprocess, lexical analysis, syntax analysis, semantic analysis, and other functions of the compiler front-end are implemented with the help of the Clang of the LLVM compiler. Based on the intermediate representation (IR) generated by the Clang, the front end of the SDPNM compiler completes data flow analysis, memory mapping, code optimization, and intermediate code generation. Data dependency within and between the loops are analyzed, and the data flow graph (DFG) and inter-loop data dependency analysis (DDA) required by the middle end of the compiler are generated respectively. DFG describes the operations in the same iteration in a loop body and the data dependencies between them, while DDA describes the data dependencies between different loops or operations in different iterations in a loop body. To reduce the frequency of communication between PEAs and GPP or the frequency of shared memory access, the SDPNM compiler introduces inter-loop DDA. By analyzing the source of input data of the loop, the data dependencies between loops and iterations within the loop are obtained. The dependency relationship between the input of the current loop and the output of other loops is analyzed. Combined with the analysis results of inter-loop DDA, the communication channels between the loops are determined. This solution ensures that as many low-cost interconnections as possible are used to improve the overall performance. Algorithm 1 is a part of the inter-loop DDA result. It shows that the data of load73 to load94 in loop8 depends on loop7, and this part of communication can be realized by using the interconnection between PEs instead of shared memory.

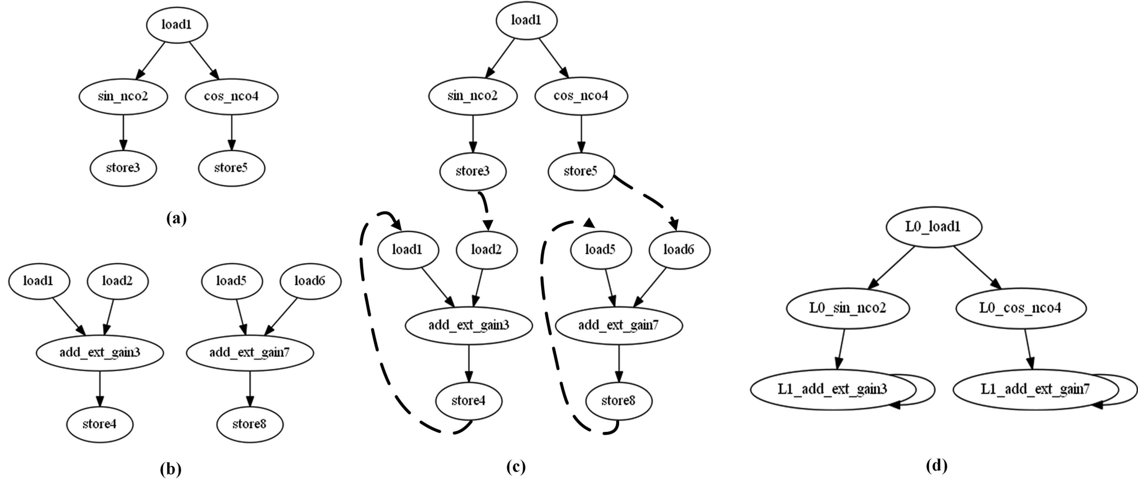


**Figure 8** Example of the basic block merge process. (a) Loop0 DFG; (b) loop1 DFG; (c) initial merge of DFGs; (d) DFG after middle-end optimization.

### 3.3 Middle-end

For the compiler middle-end, machine-independent optimizations of IRs, such as constant propagation, and dead code elimination, are made to reduce the complexity of back-end compilation. In the meanwhile, optimizations of IR based on the structure of PEAs are also made to improve the quality of machine code generated by the compiler. The optimization process of the middle-end compiler can be divided into two parts: legalization and basic block merge. The operation nodes in DFGs from the compiler front end normally are not fully consistent with the operations the PEs can support. In the meanwhile, as described in Section 2, the operations supported by PEs are customized for domain acceleration and a PE node can support several operation nodes in DFGs. To exploit the performance potential of PEAs, the compiler is designed to support the corresponding instructions set. Combining the characteristics of data flow and PE operations, the legalized DFGs tailored for the instruction set and the hardware architecture of SDPNM are generated through the instruction legalization process. In this way, the nodes in DFGs are compatible with the actual hardware structure.

The second step is for full utilization of the interconnections which are the advantages of the hardware resource of SDPNM as the interconnections are more cost-effective compared with shared memory for inter-loop data dependencies. To reduce the communication between PEs via shared memory, several independent basic DFG blocks are merged into hyper DFGs which describe the operators and data dependency among them based on the DDA obtained from the compiler front-end. This is essential for modeling in back-end compilation. Figure 8 shows an example of a basic block merge process. Figures 8(a) and (b) are DFGs for two independent loops: loop0 and loop1. The input of the algorithm includes these DFGs and the inter-loop DDA from the compiler front end. Figure 8(c) shows that these independent DFGs are merged into one DFG directly considering the inter-loop DDA shown in Algorithm 2. Then the vertices involving load and store operations are removed and relevant edges are modified. For example, the data sent to memory by operations store3 and store5 in loop0 are loaded from memory by two operations load2 and load6 in loop1. An edge is added from operation sin_nco2 in loop0 to add_ext_gain3, and a similar edge between cos_nco4 in loop0 and add_ext_gain7 is also added by compiler middle-end. In addition, the data loaded by the operation load1 in loop1 is from the operation store4 in the previous iteration. An edge from add_ext_gain3 to add_ext_gain3 in loop1 is added, and so is the edge from add_ext_gain7 to add_ext_gain7. Figure 8(d) is the DFG after optimization, and no information from the front end is lost. However, the communication between PEAs and GPPs, as well as that between PEs via shared memory, is significantly reduced and the throughput is improved.

### 3.4 Back-end

The back-end compiler involves mapping and translation. Mapping means finding the corresponding relation between the optimized IR operations along with their dependencies and the hardware structure of SDPNM. The operations are mapped to the computing resources PEs while the data dependencies

**Algorithm 2** Inter-loop DDA of loop0 and loop1

```
1: ### loop0 ###
2: − − − innerloop = 1 − − −
3: SM → L0.I1 load1 (adc_data + 4)
4: − − − innerloop = 2 − − −
5: SM → L0.I2 load1 (adc_data + 5)
6: − − − innerloop = 3 − − −
7: SM → L0.I3 load1 (adc_data + 6)
8: − − − innerloop = 4 − − −
9: SM → L0.I4 load1 (adc_data + 7)
10: ### loop1 ###
11: − − − innerloop = 1 − − −
12: SM → L1.I1 load1 (cic_sin_add + 3)
13: L0.I1 store3 → L1.I1 load2 (cic_sin_add + 4)
14: SM → L1.I1 load5 (cic_cos_add + 3)
15: L0.I1 store5 → L1.I1 load6 (cic_cos_add + 4)
16: − − − innerloop = 2 − − −
17: L1.I1 store4 → L1.I2 load1 (cic_sin_add + 4)
18: L0.I2 store3 → L1.I2 load2 (cic_sin_add + 5)
19: L1.I1 store8 → L1.I2 load5 (cic_cos_add + 4)
20: L0.I2 store5 → L1.I2 load6 (cic_cos_add + 5)
```

are mapped onto the memory or interconnections. In this way, all functions described in the original algorithm are fully realized. The mapping algorithm is essential in the whole compilation as it determines the performance and energy consumption of the implementation. The quality of the mapping is evaluated by initiation intervals (II), which is the interval between two consecutive iterations of a loop. The smaller II means a shorter execution time for each iteration and higher throughput. The mapping problem of SDPNM can be transformed into a graph homomorphism problem. More specifically, the objective of mapping is to find a homomorphic relationship between software IR and hardware models. Among them, the software IR refers to the hyper DFGs generated in the middle end of the compilation, and the hardware model can be established in the form of a topology graph according to the hardware architecture described in Section 2. The mapping problem has been proved to be NP-complete and time cost increases exponentially as the scale of PEAs increases [18]. The modulo scheduling algorithm and integer linear programming (ILP) model are used to find the mapping relation between software IR and hardware model. The main idea of modulo scheduling is to rearrange the operators in a loop and initialize the second loop within as short II as possible under the constraints that the data dependencies are not violated. Then the third loop is followed after II until the steady state is established. After the operator scheduling, the modulo machine cycle $\mathrm{mt_{op}}$ of each operator op in hyper DFG is then obtained. After the mapping model is constructed, there are commercial solvers such as Gurobi, and COPT and open-source solvers such as SCIP, and Leaves available to find the optimal solutions for ILP models. The mapping results between the hyper DFG and hardware topology can then be obtained.

The mapping results are then translated into binary machine code that can be executed on SDPNM, in other words, the configuration contexts. To reduce the cost of manual configuration, the compiler supports programming methods similar to assembly (hereinafter referred to as configuration packages). The translation process can be briefly divided into two steps: from the mapping results to configuration packages, and then to binary machine code. Configuration packages contain critical information describing each PE's behavior in every machine cycle, including the type of operation, the source of the input, and the destination of the output. The type of operation is obtained by analyzing the mapping results of operators while input and output are determined by analyzing the dependencies in the hyper DFG. If some certain dependency is mapped on shared memory or global registers, the translator will allocate it to the resource with the lowest cost automatically. If the interconnections between PEs are selected, the translator calculates the relative relation between the PEs. Then the source of the input and destination of the output for corresponding configurations are generated. A hash table is utilized to describe the correlation between the configuration packages and the binary machine code. This hash table is obtained by analyzing the structure of PEAs and the instruction decoders. By parsing every configuration line of each PE, binary machine code can then be generated via a simple table lookup method.

### 3.5 Evaluation of compilation tools

The compiler front end is built with C++ based on LLVM, and the middle end and back end are constructed using Python. The ILP solver used for mapping is Gurobi 9.1.2 [19]. The evaluation of the

**Table 2** Comparison with state-of-the-art compilation tools.

| | FPGA toolchain | SDCs toolchain | | | | |
|---|---|---|---|---|---|---|
| | | [20] | [21] | [22] | [23] | This work |
| Language | Verilog HDL | C language based | C language based | C language based | SPATIAL | C language based |
| Programmability | Low | High | High | High | High | High |
| Compiler | Vivado/quartus | CGRA-ME | RAMP | OpenCGRA | SARA | SDC mapper |
| Portability | Low | High | Low[a)] | Medium[b)] | Low[a)] | High |
| Compilation time | Hours | Minutes | Tens of seconds | Tens of seconds | Tens of seconds | Seconds |
| Simulator | Modelsim | Modelsim | Gem5 | Cycle-accurate Python | Cycle-accurate C | Cycle-accurate C |
| Simulation speed | Slow | Slow | Fast | Fast | Fast | Fast |

a) Specified for certain hardware architecture.
b) Modifications are required for adapting to new hardware architecture.

compiler involves both compilation time and performance. Compilation time affects the development cost of SDPNM while the performance represents the efficiency of the algorithms implemented on SDPNM. The evaluation platform is based on the Intel(R) Core(TM) i-6800 K 3.4 GHz quad-core processor. Before the performance evaluation, the RTL simulation using the binary configuration contexts obtained from the compiler shows the expected output signal, which verifies the function of the compiler.

#### 3.5.1 *Comparisons of different compilers*

Table 2 [20–23] compares two types of tool chains (FPGA and SDCs) in terms of compilation time as well as programmability, portability, and simulation time. In terms of programmability, the toolchains of FPGA are relatively low. The programming language of FPGA, hardware description language such as Verilog, are normally low-level language and paralleled programming which is very unfriendly to programmers. High-level synthesis (HLS) improves the software programmability of FPGAs but the efficiency is still not satisfactory. One-bit programming granularity makes the synthesis of FPGA require hours for simple benchmarks such as cap, conv, matmul, and h2v2. In the meanwhile, it is infeasible to adapt to other hardware architecture which means rather low portability. On the contrary, SDC toolchains, based on coarse-grained series programming granularity which is consistent with the software developer's mindset, are mostly of high programmability. For the same benchmarks, the compilation time of the SDCs toolchain is reduced to minutes and even seconds. Among the SDC toolchains, the proposed compilation tools also have advantages in compilation time in addition to portability and simulation time. In terms of portability, RAMP [21] and SARA [23] are both limited to certain designs of the hardware architecture while OpenCGRA [22] requires modifications to the compiler algorithm if adapted to new types of hardware architectures. Overall, the proposed compilation tools are well-rounded in different aspects.

#### 3.5.2 *Performance evaluation*

In addition to the comparisons of the intrinsic metrics of complies, the performance of the configuration contexts generated by the compiler is evaluated using a real DDC receiver application. There are two perspectives of evaluation: throughput and overall execution time. Throughput indicates the amount of data to be processed per unit time similar to the metric instruction per cycle (IPC) for CPU performance evaluation. The execution time directly reflects the speed of the algorithms implemented on the SDPNM. Since the DDC has several loops, the II cannot be obtained. Therefore, the throughput and the execution time of the configuration contexts generated manually and by the proposed compilation tools are compared respectively at the frequency of 100 MHz.

Since the throughput bottleneck of the whole algorithm is usually determined by the multiply and accumulation (MAC) units, therefore the number of PEs that are configured to multiply-accumulate operator are compared. Take finite impulse response as an example, the throughput for both manual and compiler methods is 3.2 Gbit/s/Array and 32 PEs are configured to MAC, although the configuration contexts are not identical. The execution time is obtained by measuring the interval between the first input data to be loaded and all the output data are generated. The execution time for both methods is 42470 ns, which means that the quality of configuration contexts automatically generated by the proposed compiler is as good as the manually generated counterpart but requires much less time.

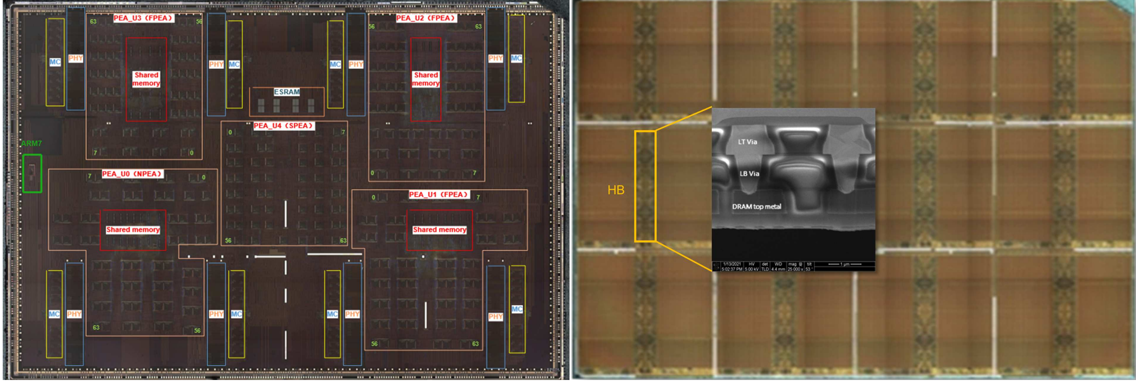


**Figure 9** (Color online) Die micrographs of logic die (left) and DRAM die (right).

**Table 3** Specification summary of logic die and DRAM die.

| | Logic die | DRAM die |
|---|---|---|
| Technology (nm) | SMIC 40 | PSMC 25 |
| Area ($mm^2$) | 16.96 × 11.87 | 16.96 × 11.87 |
| Frequency (MHz) | 200 | 200 |
| Power (W) | 0.5 | 1 |
| Voltage (V) | 1.1 (core), 3.3 (IO pad) | 1.1 |
| Capacity (GB) | – | 1 |
| Bandwidth (GB/s) | – | 287 |
| Energy (pJ/b) | – | 0.88 |

## 4 Chip implementation and evaluations

A prototype chip is fabricated, and the micrographs of the logic die and DRAM die are shown in Figure 9. The specifications of both dies are listed in Table 3. The logic die is fabricated with SMIC 40 nm 1P10M process, and the DRAM die is fabricated with PSMC 25 nm technology. The hybrid bonding is implemented by XMC. Both the logic and DRAM dies are of the same dimension of 16.96 mm × 11.87 mm for the requirement of hybrid bonding. The logic die is stacked above the DRAM die using Cu-to-Cu metal bonding. Both logic and DRAM dies operate at 200 MHz with a 1.1 V power supply. The die-to-die bandwidth can reach up to 287 GB/s. In addition, the energy efficiency of the logic-to-memory interface is 0.88 pJ/bit. The comparison with other in-memory and near-memory processing designs is listed in Table 4 [24–26]. The proposed SDPNM outperforms other designs in terms of the ratio between bandwidth and capacity as well as energy cost.

Figure 10 shows the demonstration of the evaluation board where the proposed SDPNM prototype chip is mounted in the socket. The board is powered by a 12 V DC voltage and input commands from the computer are sent to the JTAG interface of the board via a J-Link debug emulator. The output from the board is sent back to the computer via a USB to Series interface. Experimental results for both compute-intensive and data-intensive tasks are listed in Table 5 [27]. The comparisons in performance and energy efficiency with state-of-the-art FPGA (Xilinx V7 690 T, TSMC 28 nm) and a high-performance commercial heterogeneous multicore DSP+Arm System-on-chip by Texas Instrument (66AK2G12) are included. The performance of FPGA outperforms the proposed SDPNM due to the more advanced fabrication technology, higher operation frequency 250 MHz, and three times more multipliers, etc. The advantages of SDPNM are in energy efficiency, and the average ratio between the proposed architecture and the state-of-the-art FPGA is 33.1×, ranging from 8.2× to 104.1×. The DSP SoC is designed for digital signal processing algorithms and energy efficiency is better than the FPGA implementation. Still, the proposed SDPNM outperforms the DSP SoC by 10.52×, ranging from 3.08× to 18.71× in terms of energy efficiency.

**Table 4** Comparison with other in-memory and near-memory processing designs.

| | CIM SRAM [24] | 2.5D PNM HBM2 [25] | 3D TSV HBM2 [26] | This work |
|---|---|---|---|---|
| Technology | 16 nm | 1x[a)] nm/7 nm | 20 nm/20 nm | 25 nm/40 nm |
| Capacity | 4.5 Mb | 80 GB | 6 GB/cube | 1 GB |
| Bandwidth | – | 1935 GB/s | 1200 GB/s/cube | 287 GB/s |
| Bandwidth/Capacity | – | 24.2 | 200 | 287 |
| Energy | – | 4.47 pJ/bit | 2.75 pJ/bit | 0.88 pJ/bit |

a) 1x is an estimated value.

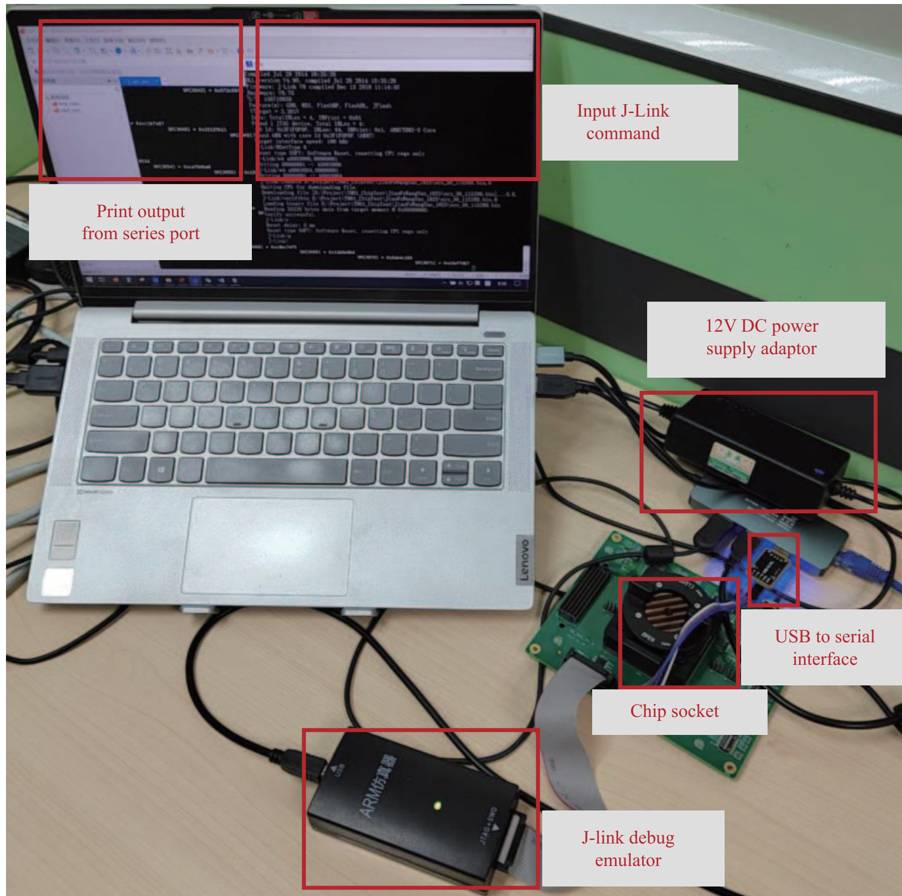


**Figure 10** (Color online) Demonstration of evaluation board with the proposed SDPNM prototype chip.

**Table 5** Performance and energy efficiency comparison with state-of-the-art FPGA and DSP.

| | Throughput (GOPS) | | | Energy efficiency (GOPS/W) | | |
|---|---|---|---|---|---|---|
| | FPGA | DSP [27] | This work | FPGA | DSP [27] | This work |
| NCO | 900 | 40 | 233 | 18 | 36 | 331 |
| CIC | 900 | 40 | 117 | 18 | 53 | 166 |
| FIR | 900 | 40 | 372 | 18 | 37 | 530 |
| FFT | 63 | 8 | 93 | 1.26 | 7 | 131 |
| GEMM | 900 | 40 | 372 | 18 | 36 | 530 |
| YOLO V3 | 900 | 40 | 104 | 18 | 48 | 148 |

## 5 Conclusion

This paper presents a software-defined process near memory computing architecture, which is a practical architecture with high efficiency and flexibility using a mature chip fabrication and bonding process. In addition to conventional reconfigurable computing elements, the memory is also included in the software-defined processing using 3D hybrid bonding integration. A compilation system is also designed to facilitate

the dynamic reconfiguration of the computing resources along with the memory access. Experimental results show that the energy efficiency of the proposed SDPNM outperforms the state-of-the-art FPGA implementations by 33.1× on average for both compute-intensive and data-intensive applications.

**Acknowledgements** This work was supported by National Key R&D Program of China (Grant No. 2021YFB3100903), Beijing Superstring Academy of Memory Technology, and National Natural Science Foundation of China (Grant Nos. 61834002, 62104129, 2022-XXXX-ZD-005-00).

**References**

1 Su L, Naffziger S. Innovation for the next decade of compute efficiency. In: Proceedings of the IEEE International Solid-State Circuits Conference, San Francisco, 2023. 8–12
2 Moore G. No exponential is forever: but "forever" can be delayed! In: Proceedings of the IEEE International Solid-State Circuits Conference, San Francisco, 2003. 20–23
3 Hennessy J L, Patterson D A. A new golden age for computer architecture. Commun ACM, 2019, 62: 48–60
4 Liu L, Zhu J, Li Z, et al. A survey of coarse-grained reconfigurable architecture and design. ACM Comput Surv, 2019, 52: 1–39
5 Wei S, Liu L, Zhu J, et al. Software Defined Chips: Volume I. Berlin: Springer, 2022
6 Gobieski G, Atli A, Mai K, et al. Snafu: an ultra-low-power, energy-minimal CGRA generation framework and architecture. In: Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA), 2021. 1027–1040
7 Liu L, Li Z, Yang C, et al. HReA: an energy-efficient embedded dynamically reconfigurable fabric for 13-dwarfs processing. IEEE Trans Circ Syst II, 2018, 65: 381–385
8 Torng C, Pan P, Ou Y, et al. Ultra-elastic CGRAs for irregular loop specialization. In: Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, 2021. 412–425
9 Dally W J, Turakhia Y, Han S. Domain-specific hardware accelerators. Commun ACM, 2020, 63: 48–57
10 Lee D U, Kim K W, Kim K W, et al. A 1.2 V 8 Gb 8-channel 128 GB/s high-bandwidth memory (HBM) stacked DRAM with effective I/O test circuits. IEEE J Solid-State Circ, 2015, 50: 191–203
11 Park M J, Lee J, Cho K, et al. A 192-Gb 12-High 896-GB/s HBM3 DRAM with a TSV auto-calibration scheme and machine-learning-based layout optimization. IEEE J Solid-State Circ, 2023, 58: 256–269
12 Sebastian A, Le Gallo M, Khaddam-Aljameh R, et al. Memory devices and applications for in-memory computing. Nat Nanotechnol, 2020, 15: 529–544
13 Cheng C D, Tiw P J, Cai Y M, et al. In-memory computing with emerging nonvolatile memory devices. Sci China Inf Sci, 2021, 64: 221402
14 Bai F, Jiang X, Wang S, et al. A stacked embedded DRAM array for LPDDR4/4X using hybrid bonding 3D integration with 34 GB/s/1 Gb 0.88 pJ/b logic-to-memory interface. In: Proceedings of the IEEE International Electron Devices (IEDM), San Francisco, 2020
15 Jiang X, Zuo F, Wang S, et al. A 1596-GB/s 48-Gb stacked embedded DRAM 384-core SoC with hybrid bonding integration. IEEE Solid-State Circ Lett, 2022, 5: 110–113
16 Niu D, Li S, Wang Y, et al. 184QPS/W 64 Mb/mm23D logic-to-DRAM hybrid bonding with process-near-memory engine for recommendation system. In: Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, 2022. 1–3
17 Lattner C, Adve V. LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization, San Jose, 2004. 75–86
18 Hamzeh M, Shrivastava A, Vrudhula S. EPIMap: using epimorphism to map applications on CGRAs. In: Proceedings of the 49th Annual Design Automation Conference, San Francisco, 2012. 1284–1291
19 Gurobi Optimization LLC. Gurobi optimization reference manual. 2021. https://docs.gurobi.com/projects/optimizer/en/11.0/
20 Chin S, Anderson J. An architecture-agnostic integer linear programming approach to CGRA mapping. In: Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, 2018. 1–6
21 Dave S, Balasubramanian M, Shrivastava A. RAMP: resource-aware mapping for CGRAs. In: Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, 2018. 1–6
22 Tan C, Xie C, Li A, et al. OpenCGRA: an open-source unified framework for modeling, testing, and evaluating CGRAs. In: Proceedings of the 38th International Conference on Computer Design (ICCD), Hartford, 2020. 381–388
23 Zhang Y, Zhang N, Zhao T, et al. SARA: scaling a reconfigurable dataflow accelerator. In: Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA), Valencia, 2021. 1041–1054
24 Jia H, Ozatay M, Tang Y, et al. A programmable neural-network inference accelerator based on scalable in-memory computing. In: Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, 2021. 236–238
25 Choquette J, Gandhi W. Nvidia A100 GPU: performance & innovation for GPU computing. In: Proceedings of the IEEE Hot Chips Symposium (HCS), Palo Alto, 2020. 1–43
26 Kwon Y, Lee S, Kwon S, et al. A 20 nm 6 GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications. In: Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, 2021. 350–351
27 Texas Instruments. Technical reference manual for 66AK2G1X multicore DSP+Arm® KeyStone II System-on-Chip(SoC). 2019. https://www.ti.com/documentviewer/66ak2g12/datasheet