HyCUBE: A CGRA with Reconfigurable Single-cycle Multi-hop Interconnect

Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra and Li-Shiuan Peh
National University of Singapore
{manupa,aditi,tulika,peh}@comp.nus.edu.sg

ABSTRACT

CGRAs are promising as accelerators due to their improved energy-efficiency compared to FPGAs. Existing CGRAs support reconfigurability for operations, but not communications because of the static neighbor-to-neighbor interconnect, leading to both performance loss and increased complexity of the compiler. In this paper, we introduce HyCUBE, a novel CGRA architecture with a reconfigurable interconnect providing single-cycle communications between distant FUs, resulting in a new formulation of the application mapping problem that leads to the design of an efficient compiler. HyCUBE achieves 1.5X and 3X better performance-per-watt compared to a CGRA with standard NoC and a CGRA with neighbor-to-neighbor connectivity, respectively.

1. INTRODUCTION

Accelerators provide significantly improved power, performance characteristics compared to general-purpose processors. Application-specific ASIC accelerators, though optimal from the power-performance viewpoint, offer little flexibility. In contrast, the reconfigurability of FPGAs along with the recent development of high-level synthesis tools have made them a popular choice as accelerators, especially when time-to-market is of the essence. However, FPGAs offer poor power and area efficiency due to the overhead of bit-level reconfigurability.

Coarse-Grained Reconfigurable Arrays (CGRAs) have emerged as a promising alternative accelerator, providing reconfigurability at word-level, thus realizing better efficiency than FPGAs [5]. Samsung Reconfigurable Processor [9], based on the CGRA architecture template ADRES [12], is one such CGRA commercially available. A CGRA consists of an array of functional units (FU), each including an ALU, a register file, and a configuration memory as shown in Fig 1. An FU is directly connected to its neighboring FUs. The FUs share a data memory that can be directly accessed by a subset of the FUs. The host processor handles data transfer to/from the data memory in the beginning/end of execution via DMA. CGRAs are ideal for acceleration of loop kernels. The operations (including memory operations) within a loop are scheduled on the FUs, while data flows are routed between dependent operations, all at compile time. The operation schedule and routing information per loop iteration are loaded into the configuration memory prior to execution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC '17, June 18-22, 2017, Austin, TX, USA

© 2017 ACM. ISBN 978-1-4503-4927-7/17/06…$15.00

DOI: http://dx.doi.org/10.1145/3061639.3062262

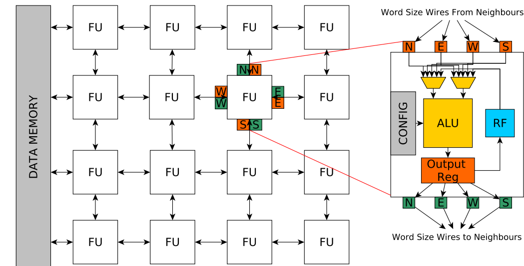


Figure 1: A 4x4 CGRA connected in a 2D mesh

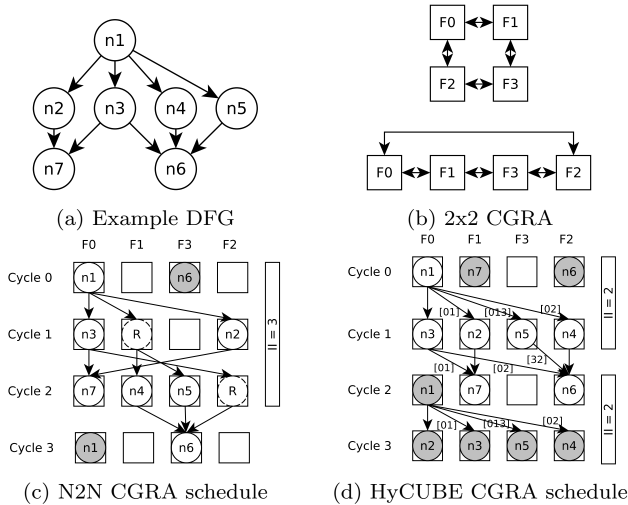
enabling per-cycle reconfiguration of the operations executing on each FU and the schedule is repeated for the number of iterations.

While CGRAs support reconfigurability through the programmable FUs, most CGRAs have static links connecting an FU only to its neighbors (Neighbor-to-Neighbor or N2N connection). Thus neighbors can be reached within a cycle, but any data transfer to a distant FU has to be routed through intermediate FUs costing multiple cycles and occupying the FU for communications, rendering it unavailable for compute. The design space explorations [2] of ADRES [12] showed that optimal energy-efficiency can be achieved through the addition of more interconnects. [15] discussed the limitations of the connectivity among FUs that resulted in the inability of the compiler to utilize distant FUs for inter-dependent operations, leading to longer schedule and lower resource utilization. It has been shown that utilization can be improved by mapping multiple loops to the CGRA [16], but at the cost of sub-optimal performance of individual loops. The N2N connection also makes the mapping of loops quite challenging for the compiler. Indeed, state-of-the-art CGRA compilers spend most of the effort in finding appropriate routes. The DRESC [13] compiler for ADRES adopts a time-consuming simulated annealing approach for routing. More recent works, such as GraphMinor [4], EPIMap [7] REGIMap [8], explore graph-based mapping that attempt to minimize routing costs and even introduce re-computation of the same operation multiple times near each of its consumer FUs simply to overcome the limitations imposed by the static interconnect of conventional CGRAs.

In this paper, we introduce a novel CGRA architecture, HyCUBE[1] that has a reconfigurable interconnect supporting single-cycle communications across distant FUs on the chip. The reconfigurable interconnect leads to a new formulation of the application mapping problem that is efficiently handled by our HyCUBE compiler. The contributions are:

- We propose a scalable novel CGRA architecture, HyCUBE combining a low-power, compiler-scheduled NoC capable

[1] We name our architecture HyCUBE as it is like the hypercube (a high dimensional cube) composed of large neighborhoods. HyCUBE enables large *virtual* neighborhoods though.



(a) Example DFG (b) 2x2 CGRA

(c) N2N CGRA schedule (d) HyCUBE CGRA schedule

Figure 2: Mapping of DFG on N2N CGRA and HyCUBE.

of delivering data across multiple hops to multiple destinations within a single cycle, creating a virtual dynamic neighborhood for the FUs.

- We provide a novel formulation of the application-mapping problem on HyCUBE, expanding the conventional modulo routing resource graph (MRRG [13]) representing the CGRA resources to include the notion of dynamic single-cycle paths and present a compiler that can generate near-optimal mappings at drastically reduced compilation time.
- We implement HyCUBE architecture using TSMC 28nm technology node. Experimental results using a range of embedded and multimedia loop kernels show that HyCUBE is, on an average, 3X better in performance-per-watt with 137% performance gain and 60% energy savings compared to a conventional CGRA.

2. MOTIVATING EXAMPLE

We first present the advantages of HyCUBE compared to the conventional CGRAs with N2N connections through a motivating example kernel. The example also illustrates the major differences in application mapping between the conventional CGRA and HyCUBE compiler. Fig 2a shows the dataflow graph (DFG) corresponding to the loop body of a kernel where the nodes represent operations and the edges represent the dependency between two operations. Fig 2c presents an optimal schedule of this kernel on a 2x2 CGRA with only N2N connection shown in Fig 2b. The operation nodes in white belong to the current loop iteration, while the shaded nodes belong to the previous or next loop iterations. Operation $n1$ is scheduled on FU: F0 in cycle 0. As F0 has two neighbors, we only have three FUs available (F0, F1, F2) to schedule the four dependent operations of $n1$ in cycle 1. Moreover, only two of these three FUs can be used to schedule operations $(n2, n3)$ because F1 (marked with Ⓡ) is used to route the data from $n1$ to the remaining dependent operations to be scheduled in cycle 2. As a consequence, $n6$ is shifted to cycle 3. In addition, F2 acts as a router in cycle 2 between $n3$ and $n6$, that are scheduled 2 cycles apart in distant FUs. According to Fig 2c, the schedule is repeated every 3 cycles, that is, a new loop iteration can be initiated every 3 cycles leading to initiation interval II=3.

Figure 2d shows an optimal schedule of the same DFG on HyCUBE that can reconfigure the interconnect at each cycle to achieve a large and dynamic neighborhood. The single-cycle multi-hop connectivity within HyCUBE enables scheduling all the dependent operations of $n1$ in cycle 1. The single-cycle, multi-hop (if necessary) path, corresponding to each data-dependence edge, is indicated within brackets. For example, [013] next to the edge $n1 \rightarrow n5$ indicates the single-cycle path from F0 (mapped with $n1$) to F3 (mapped with $n5$) via F1. This leads to a more efficient schedule with II=2 without any additional routing nodes. Note that both the edges $n1 \rightarrow n5$ and $n1 \rightarrow n2$ use the link between F0 and F1 in cycle 0 routing the same data. This is an example of the multi-cast routing ability of HyCUBE interconnect.

3. HYCUBE ARCHITECTURE

HyCUBE is a CGRA architecture where the FUs are connected in a 2D mesh topology as shown in Fig 3. The main feature of HyCUBE that distinguishes itself from previous CGRA architectures is its network. The network allows any FU to reach far-away FU (or multiple FUs) on chip within a single cycle and is completely statically scheduled. In other words, the compiler determines the configuration of the network at each cycle. HyCUBE's processing substrate consists of two types of tiles. The leftmost column (Fig 3) contains memory-operation capable tiles connected to a 4-port data memory and the rest are compute-only tiles. All the tiles comprise of an ALU, a configuration memory, and a crossbar switch. Additionally, the memory tiles contain a load-store unit (LSU) for accessing the data memory.

Network. The heart of HyCUBE's programmable interconnect is the crossbar switch. Each of the output of the crossbar switch is driven by clockless repeaters that can be configured to either let signals bypass asynchronously to the next hop (N,E,W,S tile), or to stop and receive the incoming data. This enables data to be sent across multiple tiles within a single cycle. In the event of scaling the substrate dimensions, the crossbars are not scaled since they are connected only to the neighbours. Additionally, the crossbar switch could be configured to connect the same input to many outputs, allowing a single data to be distributed to many distant destinations within the same clock cycle.

Clockless repeater links have been shown to be able to traverse up to 11 hops (across tiles that are 1mm apart, i.e., 11mm) within 1ns in a 32nm process in the SMART NoC [10]. While HyCUBE's interconnect similarly harnesses clockless repeaters, it is not a dynamically routed NoC like SMART, and is instead completely controlled by the compiler. As such, HyCUBE's interconnect comprises of only the crossbar switch and do not need any routing or flow-control logic. It also only has a single register at each port instead of buffer queues, with the register source selected at compile time. This makes for an extremely lightweight inter-

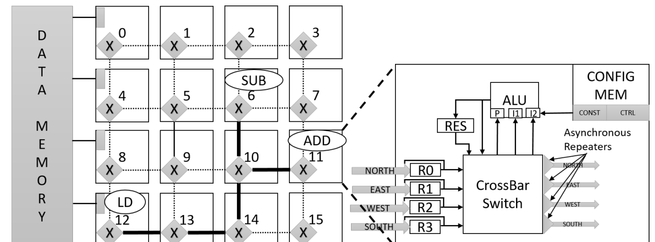


Figure 3: A 4x4 HyCUBE Architecture, with a single-cycle multi-hop multi-cast path between tiles, scheduled completely at compile time.

connect that is compatible with the stringent power budget of CGRAs. The configuration for the switch is a part of the HyCUBE instruction that is loaded each cycle from the configuration memory of each tile. This instruction essentially defines the interconnections between tiles dynamically on a cycle-by-cycle basis. To the best of our knowledge, this is the first work that introduces a CGRA capable of reconfiguring its interconnect dynamically.

Registers. The single-cycle, multi-hop interconnect has the additional benefit that the register file (RF) per tile (Fig 1) can be eliminated. In typical CGRAs, the register file is used to retain data in the current tile that will be consumed or routed away in future cycles. The addressing of the register file per cycle for reads and writes adds control overhead. Moreover, additional move operations need to be inserted to transfer data in and out of the register file for inter-dependent operations. However, in HyCUBE, the registers are moved directly into the incoming wires from each direction North, East, West and South (N,E,W,S) and HyCUBE instructions control the reads and writes to registers from/to each directional input in each cycle. The distribution of registers improve power and area efficiency.

For example (Fig 3), the LOAD operation scheduled on tile 12, needs to send the output to dependent child operations: SUB and ADD, scheduled on tile 6 and tile 11, respectively. Within a single cycle, the LOAD operation fetches the data (from the data memory) that bypasses the ALU output register RES, ejects from the crossbar towards east, bypasses registers at the input ports of tile 13 and tile 14, takes a turn towards north and enters the crossbar of tile 10 bypassing the register at the input port. From here, the data multicasts towards north and east, bypasses the registers of input ports and finally gets latched into input operand registers of the ALU (I1 or I2) of tile 6 and tile 11.

If the dependent operation (ADD or SUB) is not scheduled on the next cycle (because another operand for the operation is yet to arrive), the result of LOAD can be saved in one of the registers that were bypassed at the input of crossbars. In the cycle prior to the execution of the dependent operation, the data needs to be read from the register into the input operand registers of the ALU (I1 or I2). The input port associated with the register, remains disabled for any other communication in the cycle that the register is read. Similarly, the registers at input ports can be used to hold the data if the outgoing link is not immediately available due to contention, breaking up the single-cycle path into multiple cycles, if necessary.

Predication. The loops with control divergence form a Control Data Flow Graph (CDFG), having both control and data dependency edges. The control dependency edge is treated as a data dependency edge when the dependent instruction supports predication. HyCUBE supports such predication without requiring an explicit predicate register file. So the ALU has three input registers: predicate (P) and two operands (I1 and I2). Each operand of HyCUBE is 33-bits wide that includes an additional 1-bit predication signal. An operation can be executed only if the predicate register value is true and the embedded predicates of both input operands are also true. For example, suppose an ADD and SUB instruction are dependent upon a BRANCH instruction along the true and false path, respectively. The BRANCH output is routed to both the FUs (where ADD and SUB have been mapped) as the predicate input. This leads to execution and production of data with valid predicate for only one of them, followed by a SELECT instruction (selecting the result with the embedded valid predicate) that supplies the valid data to any further dependent instructions. The default value for the predicate input of ALU is always set to true. If the predicate data is received (probably from branch instruction executed anywhere in the substrate) from the crossbar switch, the default value is overridden, supporting partial predication in the architecture.

Configuration Memory. As illustrated in the motivating example, when loop kernels are mapped to a CGRA, the same schedule is repeated after Initiation Interval (II) cycles. This is the number of cycles between the initiation of two consecutive loop iterations in the fabric. Thus the configuration memory is required to store instructions (configurations) for II cycles. Each HyCUBE instruction encodes control information for the ALU, LSU, crossbar switch and register read/write enable signals based on the static schedule of the loop. The configuration memory also holds constants required by the operations.

4. HYCUBE COMPILER

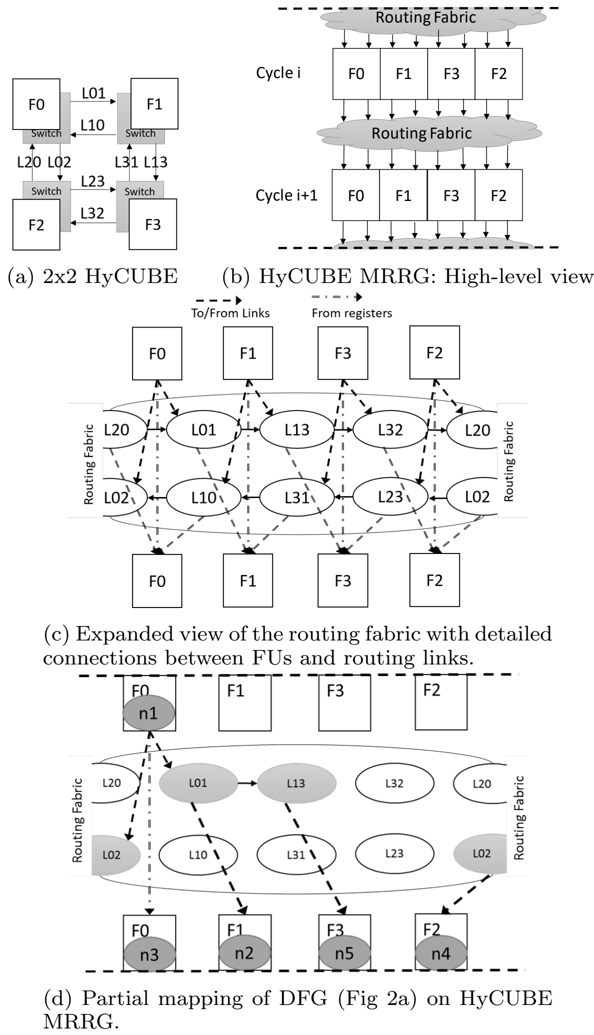
In this section, we present the HyCUBE compiler that maps application kernels onto the architecture.

4.1 Mapping Problem Formulation

Modulo Routing Resource Graph (MRRG). Given a DFG and a CGRA, the application mapping is performed through modulo scheduling where a new loop iteration can initiate execution every initiation interval (II). We first determine the lower bound on II, denoted by Minimum II (MII), as the maximum of the resource minimum II (ResMII) and recurrence minimum II (RecMII) [17]. For each II value, we create a time-extended (II cycles) resource graph of the CGRA, known as Modulo Routing Resource Graph (MRRG) [14]. As the schedule repeats after II cycles, the resources at cycle II-1 have connectivity with the resources at cycle 0 in the MRRG and hence the name "modulo". The compiler attempts to find a mapping of the DFG onto the MRRG with minimum II value.

In the MRRG for conventional CGRAs, the FUs and registers are replicated as resource nodes every cycle and the link between FU : F to its neighbor FU : F' is represented as unidirectional edge from F in cycle i to F' in cycle $(i{+}1)\ mod\ II$ for $0 \leq i \leq II-1$ [8]. The routing of a data-dependency edge, where the source and sink nodes are scheduled on distant FUs, is mapped as a multi-cycle path in the MRRG, utilizing one or more intermediate FU resource nodes for routing (e.g., $n1 \rightarrow n5$ in Figure 2c).

In contrast, HyCUBE introduces reconfigurability in the interconnect on a cycle-by-cycle basis. Thus, it additionally needs to represent the links (between FUs) as resource nodes acquired to form single-cycle, multi-hop paths. The links form a routing fabric facilitating data transfer between FUs shown in the high-level view of the MRRG with II=2 (Figure 4b) for 2x2 HyCUBE in Figure 4a. The details of the routing fabric appears in Figure 4c. The HyCUBE instance has two circular paths, [01320] and [10231], illustrated with the link resource nodes (oval nodes) constituting the paths. Each FU is connected to two link nodes in this instance and also itself (dashed edges) in consecutive cycles. Similarly, each FU has three incoming edges (dotted edges) and the computation



(a) 2x2 HyCUBE (b) HyCUBE MRRG: High-level view

(c) Expanded view of the routing fabric with detailed connections between FUs and routing links.

(d) Partial mapping of DFG (Fig 2a) on HyCUBE MRRG.

Figure 4: MRRG of HyCUBE with single-cycle routing.

in the FUs as well as the data transfer through the routing fabric can happen within a single cycle.

Problem Definition. Given a DFG $D = (V_D, E_D)$ and a HyCUBE instance, the problem is to construct a minimally time-extended MRRG of the HyCUBE instance $H_{II} = (V_H, E_H)$ that consists of two type of nodes: FUs (square nodes, V_H^F) and links (oval nodes, V_H^L) for which there exists a mapping $\phi = (\phi_V, \phi_E)$ from D:

- Operation mapping ϕ_V : each node $v \in V_D$ should have one-to-one mapping to a node $\phi_V(v) \in V_H^F$.
- Data dependence mapping ϕ_E : each edge $e_{pq} \in E_D$ (connecting nodes $v_p, v_q \in V_D$) should map to a set of links ($S_{pq} \subset V_H^L$) connecting $\phi_V(v_p)$ and $\phi_V(v_q)$.

Let us define v_r as another child of v_p and hence sibling of v_q. When selecting a set of links S_{pq} to connect $\phi_V(v_p)$ and $\phi_V(v_q)$, $S_{pq} \cap S_{pr}$ need not to be empty because the links will be carrying the same data (output of v_p) to all of its children. In the example mapping shown in Figure 4d, $\phi_V(n1)$ = (F0,0), $\phi_V(n2)$ = (F1,1) and $\phi_V(n5)$ = (F3,1) where (Fx,i) refers to FU x at cycle i. When mapping the DFG edges $n1 \rightarrow n5$ and $n1 \rightarrow n2$, the sets of links $S_{n_1n_5}$={(L_{01}, 0),(L_{13}, 0)} and $S_{n_1n_2}$={(L_{01}, 0)} are used, respectively where $S_{n_1n_5} \cap S_{n_1n_2} = \{L_{01}\}$.

4.2 Mapping Algorithm

Algorithm 1: HyCUBE Mapping Algorithm

Data: DFG, HyCUBE
Result: DFG mapped on minimally extended MRRG

1 orderedNodes = TopologicalOrder(DFG);
2 RecMII = AnalyzeRecurrenceEdges(DFG);
3 ResMII = $\frac{\#Nodes\ in\ DFG}{\#FUs\ in\ HyCUBE}$;
4 II = Max(RecMII,ResMII);
5 **while** *Mapping is not valid* **do**
6   MRRG = CreateMRRG(HyCUBE,II);
7   **foreach** *node of orderedNodes* **do**
8   **foreach** *Unmapped FU of MRRG* **do**
9   FUsWithCost.insert(FindRoutingCost(node,FU)
10   **end**
11   OptimalFU = min(FUsWithCost);
12   ScheduleAndRoute(node,OptimalFU);
13   **end**
14   II = II + 1;
15 **end**

We propose an algorithm (Algorithm 1) to find the mapping described in the problem definition guided by different cost functions. Initially, all the nodes of the DFG are sorted based on the topological order and MII is found by analyzing the DFG (lines 2-4). We incrementally increase the II value for time-extended MRRGs of the HyCUBE (lines 6-14), until a valid mapping is found between the DFG and the MRRG. For each node and for each possible unmapped FU, minimal cost (based on cost function) path to route the data dependencies from the node's parents is calculated (line 9) using $\mathcal{O}(N^2)$ Dijkstra's shortest path algorithm, where $N = |V_D|$. In the worst-case, $II = N$ (sequential execution) and number of FU nodes is $\mathcal{O}(L \times W \times II)$), where L and W are length and width of the CGRA mesh, respectively.

The cost functions play a vital role in selecting the sets of links to map DFG edges (E_D) and FUs for DFG nodes (V_D). Let us define $v_n \in V_D$ as the node currently being mapped and $v_p \in V_D$ as its parent. When mapping v_n to a new unmapped FU: $\phi_v(v_n)$, sets of links (S_{pn}) need to be selected connecting FUs $\phi_v(v_p)$ and $\phi_v(v_n)$. The choice of links and FUs are guided by the following cost functions.

Static Routing Cost (SRC) In selecting the set of links S_{pn} for $v_p \rightarrow v_n$, the number of previously unmapped links that needs to be introduced is considered as the static routing cost. The set of links (S_{pn}) may contain other links belonging to already mapped children of v_p and SRC will promote more intersection ($S_{pr} \cap S_{pn}$) with other S_{pr}, where v_r is an already mapped child of v_p. Thus, it will minimize the usage of links and preserve links for the remaining nodes.

Used Adjacent Resources Cost (UARC) Each previously unmapped link included in the set of links (S_{pn}) will have different number of adjacent resources (links/FUs) that it could connect to. UARC is the total number of used adjacent resources of each previously unmapped link. UARC discourages the use of links in congested area, where more of their adjacent resources have been consumed.

Memory Resource Cost (MRC) The memory operations could only be executed on leftmost tiles as these are the only ones connected to the memory. The use of these leftmost tiles for non-memory operations is thus discouraged, because it increases the probability of mapping failure for currently unmapped memory nodes. When selecting a memory tile $\phi_v(v_n)$ for non-memory node $v_n \in V_D$ of the DFG, a cost will be introduced that is defined as follows :
$MRC = \frac{|UnmappedMemoryNodes:\ v_m \in V(D)|}{|UnusedMemoryFUs:\ u_m \in V_H^F|}$

5. EXPERIMENTAL EVALUATION

We implemented the 4x4 HyCUBE architecture in RTL and mapped it onto TSMC 28nm process, using Synopsys Design Compiler for synthesis and Cadence Encounter for Place and Route of the design. The HyCUBE compiler is implemented as a pass in LLVM 3.9 [11] that generates HyCUBE instruction streams for loop kernel after performing the mapping based on the algorithm presented in Section 4. These instruction streams are used in our RTL simulations for estimating the power consumption. The representative loops (Table 1) are selected from MiBench [6] and CortexSuite [18]. A comparison is done against two baseline architectures: CGRA with a standard NoC (StdNoC), and CGRA with N2N Connections (N2N).

StdNoC differs from HyCUBE tile shown in Fig 5a in just one aspect: The clockless repeaters are replaced by clocked ones in the links; so data has to be latched at each hop, taking one cycle per hop. The ALU is still freed from communications, as the compiler schedules the crossbar switch and registers. Fig 5b shows the tile of a CGRA with N2N connections that has an explicit register file for storage of intermediary data. All the architectures have three registers in front of the ALU to hold predicate and input operands (P,I1,I2) and are synthesized with 4KB data memory for the entire chip and 256 Byte configuration memory per tile that can support $II <= 32$ (Mappings that have II beyond 32, will need to be partitioned). We restrict the maximum number of hops to 4 (adequate for most kernels) for 4x4 HyCUBE instance to limit the critical path delay(see Tab.2).

Performance. The previously introduced MRRG and the conventional MRRG is used when compiling for HyCUBE and N2N, respectively. The MRRG of HyCUBE had to be modified slightly, by eliminating single-cycle connectivity

Table 1: Benchmark Characteristics

Benchmark	Nodes	Edges	Domain
adpcm_dec	68	93	Telecom
adpcm_enc	88	132	Telecom
aes_enc	240	291	Security
idctflt	140	187	Video Compression
sphnix_hmm	43	70	Speech Recognition
texture_syn	57	75	Image Processing
stitch	75	102	Image Processing
fft	85	111	Signal Processing

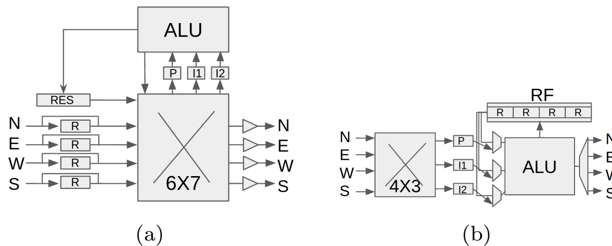


Figure 5: (a)HyCUBE tile; (b)N2N tile

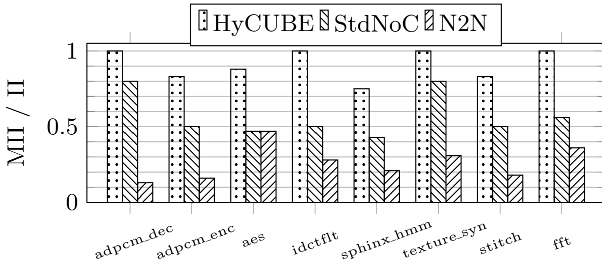


Figure 6: Quality of Mapping (MII/Mapped II)

Table 2: Critical Path Delays, Power and Area

Arch	Crit. Delay(ns)	Freq. (MHz)	Power (mW)	Area (mm^2)
N2N	0.8	1250	145.88	0.49
StdNoC	1.11	901	148.38	0.64
HyCUBE	1.42	704	115.60	0.64

between routing resources to create an MRRG for StdNoC.

As shown in Figure 6, HyCUBE has the best quality mapping for all the benchmarks and successfully achieves MII (minimum possible II) for *adpcm_dec*, *idctflt*, *texture_syn* and *fft*. The benchmarks *aes*, *sphinx_hmm* and *adpcm_enc* fail to achieve the minimum II as they have higher node count (FU constrained) or higher edge/node ratio (interconnect constrained). The benchmark *stitch* suffers as 28% of its nodes are memory operations. HyCUBE is 1.64X and 4.2X better compared to StdNoC and N2N, respectively in terms of average quality of mapping. Moreover, HyCUBE delivers quality mappings with shorter compilation time (Fig 7) due to its reconfigurable interconnect compared to N2N. StdNoC having an fixed neighbourhood interconnect, performs better compared to N2N as the network is responsible for data movement in lieu of FUs.

For further comparison of absolute performance, the different critical path timing of the three architectures are obtained after place and route PnR (Table 2). For HyCUBE, the delay depends on the maximum number of hops unlike the static delay for StdNoC and N2N. The critical path for HyCUBE goes from from the output of the ALU → crossbar → links → crossbar of the tile maximum hops away, before it is stored in a register. We restrict maximum number of hops to 4 to limit the critical path. The longer the path, the higher the capacitance of the link that needs to be driven from the repeater, stretching timing and limiting maximum frequency. Extending the maximum number of hops to 8, the entire 4x4 HyCUBE can be traversed in a single cycle, but the critical path will stretch from 1.42 to 1.59ns, and most applications do not require such distant communications. The link only contributes 10.8% of critical path delay, hence the maximum frequency at which HyCUBE and StdNoC operate do not differ much. The critical path defines the maximum frequency. Based on the critical path and II achieved for a loop L, $Time\ Per\ Iteration$ $(L)=\ MappedII_{arch}(L)\ \times Crit.Delay_{arch}$, which is the reciprocal of the throughput (loop iterations per second). Table 3 shows that HyCUBE and StdNoC are able to achieve an average throughput improvement of 137% and 101% compared to N2N, respectively (when each architecture runs at its maximum possible frequency).

Energy. HyCUBE consumes an average power of 115.60 mW at 704 MHz where crossbar switches and memories contribute to majority of the power (25.6% and 42.4%) while occupying 24% and 25.28% of total chip area, respectively. Further the ALUs, which account for 44.8% of chip area,

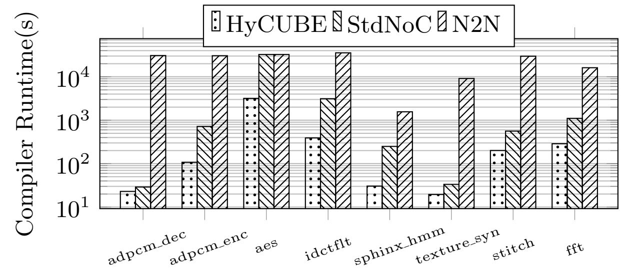


Figure 7: Compilation Time

Table 3: Throughput and Energy Comparison

Benchmark	Throughput w.r.t N2N			Energy w.r.t N2N		
	N2N	StdNoC	HyC	N2N	StdNoC	HyC
adpcm_enc	1	2.31	3	1	0.44	0.26
adpcm_dec	1	4.61	4.51	1	0.22	0.18
aes	1	0.72	1.06	1	1.41	0.75
idct	1	1.1	2	1	0.93	0.4
sphinx_hmm	1	2.02	1.97	1	0.5	0.4
stitch	1	2.31	3	1	0.44	0.26
texture_syn	1	1.87	1.83	1	0.54	0.43
fft	1	1.13	1.57	1	0.9	0.51
	Mean	**2.01**	**2.37**	**Mean**	**0.67**	**0.4**

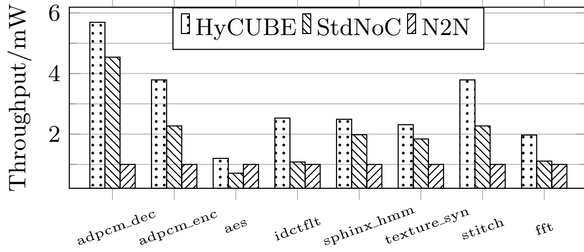


Figure 8: Performance-per-watt w.r.t. N2N CGRA

only consume 20.8% of chip power. To compare the energy consumption of an architecture for a loop L, we first derive the average power consumption (P_{arch}) of the architecture running at its maximum frequency after performing PnR, using Cadence Encounter, as shown in Table 2. Then energy per loop iteration is computed as $P_{arch} \times Time\ Per\ Iteration_{arch}\ (L)$. According to Table 3, HyCUBE and StdNoC are able to achieve a 60% and 33% average energy reduction compared to N2N, respectively, when running at their corresponding maximum frequency. The average performance-per-watt is computed based on throughput (iterations per second) and the P_{arch}. According to Figure 8, HyCUBE and StdNoC are 3X and 1.98X better compared to N2N, while HyCUBE is 1.5X better compared to StdNoC in terms of average performance-per-watt.

Comparison with other accelerators. To perform a high-level comparison against existing commercial platforms, we extracted published power and performance values of the execution of 256-FFT on ARM-Cortex A5 [1], Xilinx Artix 7 [3], Samsung Reconfigurable Processor [9]. As HyCUBE is mapped to TSMC 28nm process, all other performance numbers are scaled (1.42 times per generation due to 0.7 scaling of feature size, with the power remaining the same as core voltage is kept constant) for fair comparison. Fig 9b shows $log(Perf) = log(m) + log(Power)$ lines and the intercept of the lines indicates the logarithm of efficiency (m=Perf/Power). The Xilinx Artix 7 FPGA offers 10X performance compared to the ARM at similar performance-per-watt. On the other hand, SRP is a commercial 3X3 CGRA with N2N connections designed to achieve better power-efficiency (2X compared to ARM) but could only offer 30% performance compared to ARM. Given the reconfigurable single-cycle multi-hop interconnect, HyCUBE scales very well for 4x4 CGRA and offers 3X performance and 4X performance-per-watt compared to ARM. It is clear that CGRAs are a better choice compared to FPGAs in terms of performance-per-watt but the improvements made with regard to the interconnect of the HyCUBE enables it to improve performance while maintaining highest performance-per-watt.

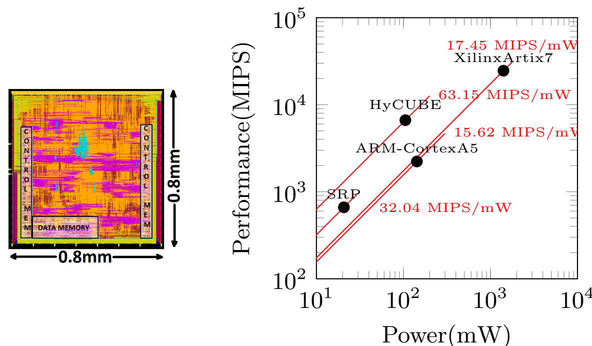


Figure 9: (a) HyCUBE chip layout (b)Power-Performance comparison with commercial processors and accelerators.

6. CONCLUSION

CGRA is a promising technology to accelerate loops. However, current reconfigurability per cycle is for the operations executing on functional units, but not for the connectivity between functional units. HyCUBE is a novel CGRA architecture that incorporates a reconfigurable single-cycle multi-hop interconnect between functional units. We introduce a novel resource abstraction model for HyCUBE that includes single-cycle routing and leads to the design of an efficient compiler. We synthesize HyCUBE architecture in 28nm process and compile real loop kernels onto the architecture. The experimental results show that average power efficiency of HyCUBE is 1.5X and 3X compared to a CGRA with standard NoC and a N2N CGRA, respectively.

7. ACKNOWLEDGMENTS

This work was partially funded by the Singapore Ministry of Education Academic Research Fund Tier 2 MOE2014-T2-2-129 and by the Singapore National Research Foundation Research Fund NRF-RSS2016-005.

8. REFERENCES

[1] Arm cortex-a5. https://goo.gl/pGytB2.
[2] Bouwens et al. Architectural exploration of the adres coarse-grained reconfigurable array. In *ARC '07*.
[3] Chen et al. Algorithmic optimizations for energy efficient throughput-oriented fft architectures on fpga. In *IGCC '14*.
[4] L. Chen et al. Graph minor approach for application mapping on cgras. *TRETS '14*.
[5] B. De Sutter et al. Coarse-grained reconfigurable array architectures. In *Handbook of signal processing systems*. '13.
[6] M. R. Guthaus et al. Mibench. In *WWC-4 '01*.
[7] M. Hamzeh et al. Epimap: using epimorphism to map applications on cgras. In *DAC '12*.
[8] M. Hamzeh et al. Regimap: register-aware application mapping on coarse-grained reconfigurable architectures. In *DAC '13*.
[9] Kim et al. Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications. In *FPT '12*.
[10] T. Krishna et al. Breaking the on-chip latency barrier using smart. In *HPCA'13*.
[11] C. Lattner et al. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04*.
[12] B. Mei et al. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *FPL '03*.
[13] B. Mei et al. Dresc: A retargetable compiler for coarse-grained reconfigurable architectures. In *FPT '02*.
[14] B. Mei et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling. *IEE-Computers and Digital Techniques '03*.
[15] Park et al. Efficient performance scaling of future cgras for mobile applications. In *FPT '12*.
[16] H. Park et al. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *MICRO '09*.
[17] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *MICRO '94*.
[18] S. Thomas et al. Cortexsuite. In *IISWC '14*.