Cayman: Custom Accelerator Generation with Control Flow and Data Access Optimization

Youwei Xiao†, Fan Cui†, Zizhang Luo†, Weijie Peng†, and Yun Liang*†
†*School of Integrated Circuits, Peking University,* *Institute of Electronic Design Automation, Peking University*
shallwe@pku.edu.cn, pku_cf@stu.pku.edu.cn, {semiwaker, weijiepeng, ericlyun}@pku.edu.cn

2025 62nd ACM/IEEE Design Automation Conference (DAC) | 979-8-3315-0304-8/25/$31.00 ©2025 IEEE | DOI: 10.1109/DAC63849.2025.11132875

***Abstract*—Custom accelerators enhance System-on-Chips' performance through hardware specialization. High-level synthesis (HLS) can automatically synthesize accelerators for given kernels but requires manual selection and extraction of kernels from applications. This paper proposes Cayman, the first end-to-end framework to synthesize high-performance custom accelerators with both control flow and data access optimization. Cayman automatically selects kernels for hardware acceleration based on a hierarchical program representation, which captures kernel candidates with general control flows. Besides, Cayman optimizes accelerators with specialized processor-accelerator interfaces for data access acceleration. Cayman further introduces a novel accelerator merging mechanism to synthesize reusable accelerators. Experiments on various benchmarks demonstrate that Cayman outperforms two state-of-the-art frameworks by 8.0× and 14.4×.**

I. Introduction

Integrating accelerators has emerged as a critical strategy for augmenting the performance of modern applications. Specialized tasks are offloaded from the CPU core to accelerators for faster execution. Customizing accelerators encompasses two principal processes: *candidate selection* decides on the offloaded programs regions for acceleration, and *hardware design* implements the hardware accelerator. Manual accelerator design [8] is inefficient owing to the inherent complexity of applications and the intricacy of hardware design. Candidate selection frameworks [29], [30] only explore candidate regions based on accelerator models without synthesizing hardware. High-level synthesis (HLS) tools [11], [15], [25], [26] can synthesize accelerator hardware design from software descriptions. Accelerators' performance, power, and area can be further optimized via design space exploration [20], [27], [28]. However, HLS frameworks rely on designers to manually select the candidate for acceleration, which is time-consuming and often leads to sub-optimal solutions for a complex application.

Prior works attempted to develop an end-to-end flow to synthesize custom accelerators. However, they have severe limitations: they either exclude control flow and data access from accelerated kernels or only synthesize sequential and unoptimized implementation, inhibiting performance improvement. Specifically, custom functional unit synthesis frameworks [2], [5], [6], [10], [16], [19], [21] only accelerate dataflow graphs (DFGs) from target application programs. In other words, they do not accelerate code regions including any control flow or data access to memory. Off-core accelerator synthesis frameworks [4], [22], [23], [31] synthesize accelerators with control flow and memory access support. However, they solely synthesize sequential control flow implementation and adopt slow processor-accelerator interfaces for data access. As a result, they fail to achieve satisfying performance speedup.

This paper proposes Cayman, the first end-to-end framework that synthesizes high-performance custom accelerators with full control flow and data access support. Cayman automates both candidate selection and hardware synthesis. Specifically, Cayman selects kernels to synthesize based on the whole-application program structure tree (wPST) representation, which captures program regions of rich control flows for hardware acceleration. Cayman introduces a dynamic programming-based algorithm with heuristic pruning and solution filtering for efficient and comprehensive selection exploration. To synthesize high-performance accelerators for selected kernels, Cayman proposes an accelerator model that considers specialized processor-accelerator data access interfaces: *coupled*, *decoupled*, and *scratchpad*. The model efficiently generates accelerator configurations comprising control flow optimization and data access interface application and conducts performance and area estimation. Moreover, Cayman proposes a novel accelerator merging mechanism that shares reconfigurable datapath units within a reusable accelerator to accelerate multiple program regions even with diverse control flows, effectively reducing area overhead.

In summary, this paper makes the following contributions:

- We introduce Cayman, an end-to-end framework to synthesize high-performance custom accelerator automatically.
- We propose a dynamic programming-based candidate selection algorithm with heuristic pruning, modeling processor-accelerator data access interfaces and control flows.
- We propose a novel accelerator merging mechanism to synthesize reusable accelerators.

For evaluation, we compare Cayman with two state-of-the-art custom accelerator synthesis frameworks on various applications. Experimental results show that Cayman outperforms NOVIA [21] and QsCores [23] by 14.4× and 8.0× with the same total area budget for the synthesized accelerators.

II. Background and Related Work

A. High-level Synthesis

High-level synthesis (HLS) [9] can synthesize hardware accelerators from software kernels. With a specified target clock period, HLS tools [15], [25], [26] schedule the operations to

* Corresponding author

TABLE I: Comparison between prior works and Cayman.

Methods	design entry	candidate selection	control flow	data access	hardware sharing
HLS[1]	kernel	manual	optimized	specified	/
CFU[2]	application	auto	/	scalar-only	restricted
OCA[3]	application	auto	sequential	slow	restricted
Cayman	**application**	**auto**	**optimized**	**specialized**	**flexible**

[1] [3], [11]–[13], [15], [17], [18], [24]–[26]; [2] [2], [5], [6], [10], [19], [21]; [3] [4], [22], [23], [31]

cycles within the kernel, subsequently crafting the accelerator design based on the scheduling outcome. HLS tools support various optimization techniques, such as loop pipelining and unrolling, which may transform the kernel's source code or direct the scheduler to refine the control logic of the accelerator, thereby enhancing performance. DSE frameworks [20], [27], [28] are proposed to explore the optimizations automatically.

Limitations: As shown in Table I, HLS tools' design entry is software kernels, requiring designers to extract and specify manually, rather than selecting them from applications automatically. As a result, HLS requires significant human interaction. The same problem applies to ASSIST [17], Longnail [18], and Bluespec Accelerate-HLS [3], all of which adopt HLS but require manual candidate selection. Besides, designers must choose data access interfaces for external memory operations in kernels, which impact scheduling and overall performance but are overlooked by existing DSE frameworks.

B. Prior Works on Custom Accelerator Synthesis

Based on the types of synthesized accelerators, we segregate prior custom acceleration synthesis works into two classes: custom functional unit (CFU) synthesis frameworks [2], [5], [6], [10], [19], [21] and off-core accelerator (OCA) synthesis frameworks [4], [22], [23], [31].

Limitations: As shown in Table I, CFU synthesis frameworks select candidates exclusively in the DFG form, precluding control flows. Their data access interfaces only take scalar operands and return results since they do not support other memory accesses. OCA synthesis frameworks support control flow and memory access. However, they only implement sequential control logic and show data access interfaces, such as the scan-chain interface characterized by high latency and low bandwidth in [22], [23], significantly circumscribing performance gains. For hardware sharing, some CFU synthesis frameworks [5], [21] merge candidates into a reusable CFU, while some OCA frameworks [4], [23] endorse sharing solely among almost identical loops or functions. In other words, they support restricted hardware sharing and cannot reuse one accelerator to accelerate multiple kernels with diverse control flows, missing area-saving opportunities.

III. Methodology

A. Overview

As overviewed in Fig. 1, Cayman ingests target application programs as input and automatically identifies the high-performance solutions, each comprising multiple accelerators. The framework compiles programs into LLVM IR and builds

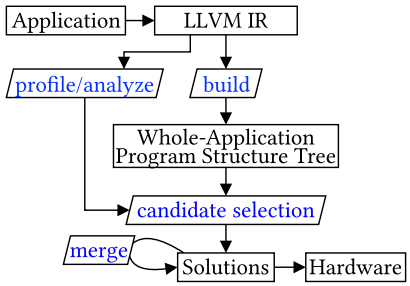
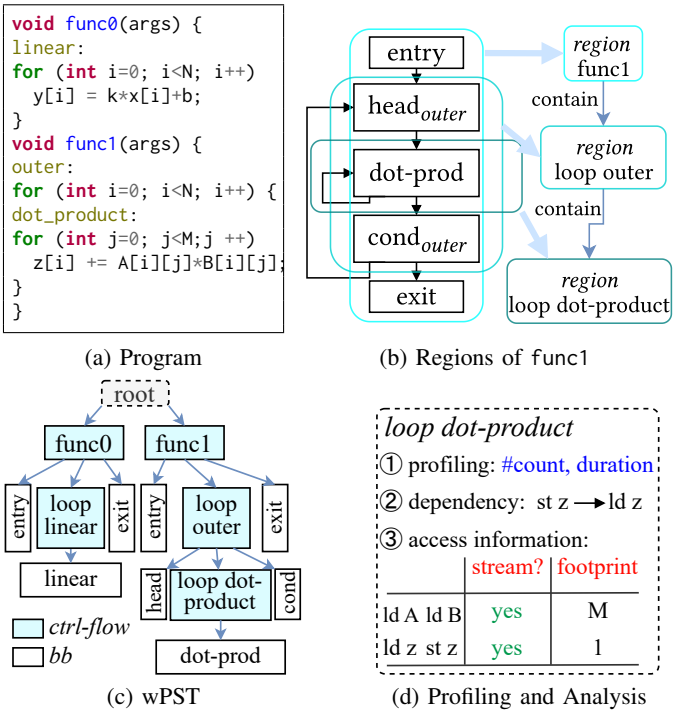


Fig. 1: Overview of the Cayman framework



(a) Program (b) Regions of func1 (c) wPST (d) Profiling and Analysis

Fig. 2: Example

the whole-application program structure tree (wPST) representation. The framework also runs profiling and program analysis through LLVM passes. The framework then leverages the profiling and analysis outcomes to conduct candidate selection on the wPST and produces solutions comprising selected kernels and accelerator configurations.

B. Representation, Profiling, and Analysis

Cayman introduces the whole-application program structure tree (wPST) representation for candidate selection. Traditional program structure tree (PST) [14] identifies single-entry-single-exit (SESE) *regions* inside a function as vertices and represents the containing relationships between regions as edges. The wPST extends PST with a *root* vertex representing the entire application and multiple function vertices representing every included function. We consider only SESE regions as legal acceleration candidates because the SESE property is necessary to guarantee that the candidate's execution can be completely isolated from the main processor through offloading and synchronization at their entry and exit points, respectively. The wPST captures all feasible acceleration candidates as its region vertices in a target application. Specifically, the

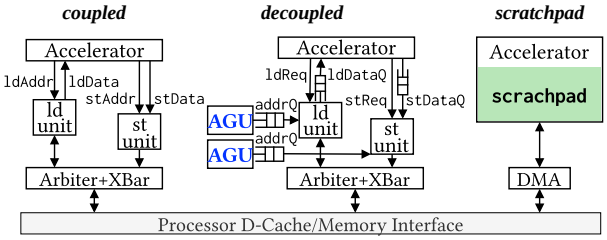


Fig. 3: CPU-accelerator data communication interfaces

wPST represents two region types: *bb* regions, corresponding to basic blocks, and *ctrl-flow* regions, corresponding to general control flow including loops and conditionals.

Cayman profiles application execution information including duration and execution count for every program region. The profiling results are essential for locating hotspot kernels and estimating performance gains. Besides, Cayman conducts program analysis to collect data access information, including memory dependencies, memory access patterns, and access footprints. Specifically, Cayman identifies loop-carried dependencies for every loop region, identifies *stream* access patterns whose access address sequence can be statically computed, and analyzes the access address range for every memory operation. The data access information is crucial for accelerator configuration decisions, including control flow optimizations and data access interface selections.

The program in Fig. 2a contains two functions, and Fig. 2c presents the corresponding wPST, showcasing *ctrl-flow* and *bb* regions as candidates for acceleration. Specifically, Fig. 2b shows the control-flow graph (CFG) of `func1` and illustrates three regions comprising control flows. Fig. 2d presents profiling ① and analysis ②③ results for the "loop dot-product" region. It includes one loop-carried dependency between the store and load operations to/from `z[i]`. Besides, all access operations have the *stream* access pattern, while `ld A` and `ld B` have the access footprint of `M` according to the induction variable `j` and `ld z` and `st z` have the access footprint of `1` due to the invariant `i` in the loop.

C. Accelerator Modeling

Data access interfaces between the processor and accelerators have nonnegligible impacts on the performance and area usage of the accelerators. However, it is overlooked by prior exploration works. Cayman systematically models three types of data access interfaces: *coupled*, *decoupled*, and *scratchpad*, as shown in Fig. 3:

- *Coupled* interface uses load/store units for access. A load-/store unit initiates data access to the memory system upon receiving requests with a target address from the accelerator, causing the accelerator to stall until the response arrives.
- *Decoupled* interface allocates dedicated address generation units (AGUs) for load/store units. The AGUs compute access addresses independently from the accelerator, which enables load units to start data access earlier than the accelerator's requests and enables store units to send out data later, both reducing the accelerator stall cycles. However, the *decoupled* interfaces pay extra area overhead for AGUs and data buffering FIFOs, and are only applicable to *stream*-pattern accesses for the requirements of independent address generation.
- *Scratchpad* interface reserves a dedicated buffer within the accelerator to cache a memory range. Data synchronization between the scratchpad and the memory system occurs before and after accelerator execution through a direct memory access (DMA) engine. It is beneficial when most positions in the cached memory range are accessed multiple times. However, the *scratchpad* interface incurs prominent area overhead for the scratchpad buffer and the DMA engine and requires statically analyzed footprints to determine the scratchpad size.

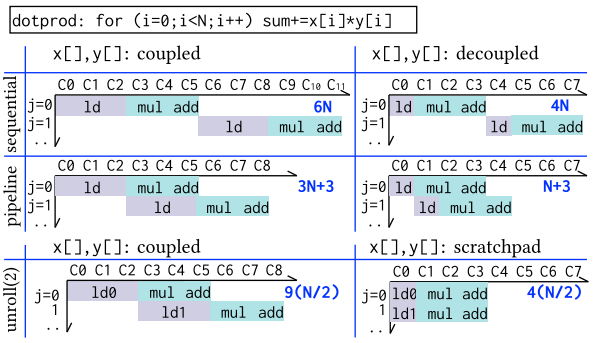


Fig. 4: Impacts of data communication interfaces.

Fig. 4 exemplifies data access interfaces' impacts on accelerator performance considering different control flow implementations. For sequential loop, the *coupled* interface leads to latency of 6N cycles, where N is the tripcount of the loop. The *decoupled* interface optimizes the latency to 4N cycles since it reduces the stall cycles associated with the loads by starting loads sooner before the accelerator requests. For loop pipelining, the *coupled* interface gives a pipeline initial interval (II) of 3 cycles, while the *decoupled* interface achieves an II of 1 cycle. For loop unrolling with the factor 2, the *coupled* interface leads to a latency of 9(N/2) cycles, and the *scratchpad* interface optimizes the latency to 4(N/2) cycles by enabling parallel access to the scratchpad without stalls. Cayman considers interface specialization for different access operations to achieve full data access acceleration.

Accelerator Configuration: Given the selected kernels for acceleration, Cayman can synthesize multiple accelerator configurations comprising control flow optimizations like loop unrolling and pipelining and data access interface strategies to achieve performance-area tradeoffs. Given the huge design space, Cayman adopts a fast exploration strategy. Specifically, it tries unrolling loops without loop-carried dependencies and pipelining the innermost loops after unrolling. Subsequently, Cayman determines data access interfaces heuristically for memory access operations: use *scratchpad* interface for operations with the total access count β times larger than the access footprint, indicating the necessity for caching; use *decoupled* interface for operations inside pipelined loops to achieve ideal II; and otherwise, use the *coupled* interface for area saving. Memory partitioning is configured for *scratchpad* interfaces inside unrolled loops to support parallel access. Consequently,

Algorithm 1: Candidate Selection

```
   Data: wPST T, profiling/analysis results R
   Result: Pareto-optimal selections
1  Function DP(vertex v):
2      if prune(v, R) then return ;
3      if v is bb then
4          F[v] ← filter(pareto(accel(v,R)));
5      else
6          F[v] ← ∅;
7          for u ∈ v.children do
8              DP(u);
9              F[v] ← filter(F[v] ⊗ F[u]);
10         if v is ctrl-flow then
11             F[v] ← filter(F[v] ∪ pareto(accel(v,R)));
12 DP(T.root);
13 return F[T.root];
```

Cayman efficiently generates candidate accelerator configurations with optimized control flow and data access.

Performance and Area Estimation: Cayman's accelerator model estimates performance gains and area usage for selected kernels with accelerator configurations without synthesizing complete hardware. The estimation comprises three steps: (1) apply loop unrolling to transform the kernels according to the configuration; (2) only synthesize pipelined loop regions P and sequential basic block regions B; and (3) estimate accelerators' latency and area through a bottom-up approach. Specifically, Cayman considers diverse interface-specific latencies and resource constraints when scheduling data access operations during synthesis. For every region in $P \cup B$, the cycle count is computed based on the scheduled latency and profiled execution count, and the area is retrieved from the synthesis report. For an outer region, the cycle count is calculated by summing the cycle counts of its child regions, and the area is estimated as the sum of its child regions' area plus extra control logic overhead. By doing so, the total cycle count of the accelerators is efficiently estimated as $Cycle_{cand}$. Given the profiled total duration of the accelerated kernels T_{cand} and the original execution duration of the entire program T_{all}, the overall performance gain is estimated as

$$Speedup = T_{all}/(T_{all} - T_{cand} + Cycle_{cand}/F) \quad (1)$$

where F is the target frequency of the synthesized accelerators.

D. Candidate Selection

Candidate selection determines a set of kernels corresponding to non-overlapping wPST subtrees for hardware acceleration to achieve the most performance gains under specific area budgets. Cayman models candidate selection as a knapsack problem: every wPST region, either *ctrl-flow* or *bb*, corresponds to a knapsack item, and the performance gain and area usage of the synthesized accelerator are the item's profit and weight, respectively. The knapsack problem has the constraint: if a vertex is selected, all its descendants cannot be selected, avoiding overlapping acceleration of kernels.

Cayman proposes a dynamic programming approach to solve the knapsack problem, as shown in Algorithm 1. The function DP(v) (line 1) solves the knapsack problem considering regions contained in the wPST subtree rooted at the vertex v. The function prune (line 2) terminates the search if the region v is identified as not a hotspot worth acceleration according to the profiling and analysis results R and a heuristic pruning strategy. We define $F[v]$ as the area-performance Pareto-optimal solution sequence accelerating kernels from v's subtree. Every solution $\phi \in F[v]$ comprises one or more non-overlapping kernels with accelerator configurations. Algorithm 1 builds $F[v]$ (line 3-11) according to the following recursion formula:

$$F[v] \leftarrow \begin{cases} \texttt{pareto(accel}(v,R)\texttt{)} & \text{if } v \text{ is } bb \\ \texttt{pareto(accel}(v,R)\texttt{)} \cup \bigotimes_{u \in v.children} F[u]) & \text{if } v \text{ is } ctrl\text{-}flow \\ \texttt{pareto}(\bigotimes_{u \in v.children} F[u]) & \text{otherwise} \end{cases}$$

where accel calls Cayman's accelerator model to generate configurations for selected kernels, producing accelerators with optimized control flow and data access and estimating performance and area as discussed in Section III-C, and pareto produces a Pareto-optimal sequence for the given solutions. Specifically, there are three cases for $F[v]$'s construction according to the type of vertex v. If v is a *bb* region, $F[v]$ gets accel(v, R) solutions accelerating v's dataflow subgraphs. If v is a *ctrl-flow* region, either the region v gets accelerated as an extracted kernel by a single accelerator, corresponding to accel(v, R) solutions, or its descendant regions get accelerated separately, applying the $\otimes$ operation to combine solutions from sibling subtrees rooted at v's children: $F[u_1] \otimes F[u_2]$=pareto$(\{\phi_1 \cup \phi_2 \mid \phi_1 \in F[u_1], \phi_2 \in F[u_2]\})$. Otherwise, v is the *root* vertex that cannot be selected directly, and $F[T.root]$ applies the $\otimes$ operation to combine solutions from different functions to form the overall selection solutions.

Algorithm 1 also features the filter function (line 4,9,11) that filters out unnecessary solutions that are too close to each other. Specifically, for a Pareto-optimal solution sequence $\{\ldots, \phi_i, \phi_{i+1}, \ldots, \phi_j, \ldots\}$ with increasing area $a_{i+1} > a_i$, suppose ϕ_j is the first solution whose area a_j is α times larger than that of ϕ_i. That is, $a_j > \alpha a_i$, filter removes solutions $\{\phi_{i+1}, \ldots, \phi_{j-1}\}$ from the sequence. After filtering, every neighboring solution pair (ϕ_i, ϕ_{i+1}) satisfies that $a_{i+1} > \alpha a_i$. For any knapsack with an area limit A, filter reduces the maximum possible number of solutions from A to $log_\alpha A$, greatly improving selection efficiency.

Algorithm Complexity: The complexity of Algorithm 1 is $O(N \log_\alpha^2 A + E)$, where N denotes the number of unpruned vertices in the wPST, A denotes the discretized area limit, and E denotes the total time complexity of the accelerator model. The term $N \log_\alpha^2 A$ arises from $F[v]$'s construction, where the $\otimes$ operation enumerates solution combinations from two F sequences, each of which contains at most $\log_\alpha A$ solutions.

E. Accelerator Merging

After the candidate selection, Cayman performs accelerator merging, enabling multiple program regions with distinct control flows to share a reusable accelerator. The core strategy is to merge basic blocks that share common operations by inserting multiplexers with reconfiguration bit registers, producing a reconfigurable datapath unit. Cayman then combines candidates that include the merged basic blocks into a reusable accelerator, each maintaining a standalone finite-state machine (FSM)

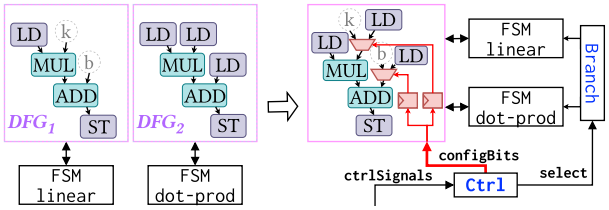


Fig. 5: Accelerator merging

for control. Upon triggering the reusable accelerator, a global control unit, named *Ctrl*, transmits configuration data to the datapath units and selects the appropriate control logic FSM to trigger. This merging strategy is based on the observation that basic blocks housing operations like floating-point arithmetic and data access contribute significantly to the accelerator area, while control logic FSMs demand much less area. Fig. 5 presents the example of merging the loop linear and loop dot-product candidates from Fig. 2. Cayman synthesizes a reusable accelerator containing a reconfigurable hardware unit merged from DFG_1 and DFG_2. The reusable accelerator accelerates both the loops with individual FSMs.

Cayman conducts accelerator merging in a heuristic manner for every Pareto-optimal selection solution that comprises multiple accelerators. It estimates the area savings for merging every pair of basic blocks contained in the solution, merges the basic block pair with the maximum estimated area savings into a reconfigurable datapath, and combines the two accelerators containing the merged basic blocks into a reusable accelerator. The merged datapaths and accelerators are also treated as normal basic blocks and accelerators, enabling further merging with others. The heuristic merging process is repeated until no area savings can be achieved.

F. Implementation

We implement Cayman based on LLVM 18 for candidate selection, accelerator merging, and hardware synthesis. We utilize the *RegionInfoAnalysis* pass to identify regions and build wPSTs. For profiling, we implement an instrumentation pass to insert execution count and timestamp recording instructions within regions; the modified bitcode is compiled and executed to gather profiling data. For program analysis, we employ the *MemoryDependenceAnalysis* pass to get memory dependencies, implement a custom pass to identify *stream* patterns, and run the *ScalarEvolution* pass to analyze footprints. Cayman's accelerator model retrieves delay and area of operations and data access interfaces by synthesizing them with OpenROAD [1] targeting Nangate45 PDK.

IV. Evaluation

A. Experiment Setup

All applications are compiled with the "-O3" flag. The target frequency of accelerators is set to 500 MHz. Speedup is calculated as per Equation 1. Area is retrieved from reports of synthesizing accelerators into complete hardware and is presented as ratios to that of a CVA6 RISC-V tile [32]. We compare Cayman to two state-of-the-art automated accelerator

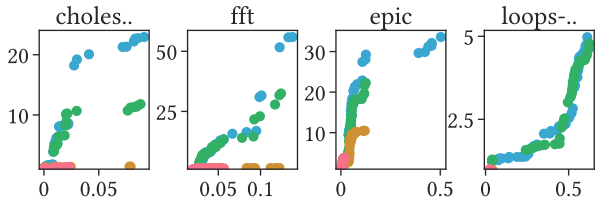


Fig. 6: Speedup (y-axis) and area (x-axis) of solutions: ● NOVIA, ● QsCores, ● *coupled*-only Cayman, and ● full Cayman.

synthesis frameworks, NOVIA [21] and QsCores [23]. We use diverse benchmarks from benchmark suites including PolyBench, MachSuite, MediaBench, and CoreMark-Pro for a comprehensive evaluation.

B. Result Analysis

We evaluate two different area budgets for the synthesized accelerators: small (25%) and large (65%). The area budgets are within the typical area range of custom accelerators [**?**], [7]. Table II presents experimental results. Cayman outperforms NOVIA and QsCores for all benchmarks. It achieves an average speedup of 14.4× and 8.0× over NOVIA and QsCores, respectively, under the budget 25%. The speedup increases to 27.2× and 15.0× when the budget is 65%, demonstrating the better acceleration potentials of Cayman. Cayman achieves superior performance because it optimizes control flow and adopts fast data access interfaces like *decoupled* and *scratchpad*. On the contrary, NOVIA fails to support control flow and memory accesses, and QsCores synthesizes sequential control flow and slow access interfaces, achieving much less performance improvement than Cayman.

Table II also presents kernel selection and accelerator configuration details of Cayman solutions. For applications with even-distributed hotspots, Cayman selects more kernels for acceleration when a larger area budget is available. One notable example is the loops-all-mid-10k-sp workload, where Cayman accelerates 13 sequential basic blocks and 1 pipelined region for the budget 25%, and accelerates 122 sequential basic blocks and 40 pipelined regions for the budget 65%. For applications with centralized hotspots, Cayman accelerates the performance-critical regions intently with more area. For example, Cayman synthesizes accelerators with the same #SB and #PR for two area budgets for 13 of 16 benchmarks from PolyBench in Table II, indicating consistent regions for acceleration. Cayman accelerates more access operations with a larger area budget: on average, it adopts 40 interfaces for the budget 25%, which is increased to 53 for the budget 65%. Besides, *decoupled* and *scratchpad* interfaces are widely adopted, occupying 83% and 81% on average for two budgets, respectively, demonstrating the significance of Cayman's interface specialization. Besides, more *scratchpad* interfaces are used when more area is available since the expensive *scratchpad* interface enables greater access parallelism.

Table II also shows that Cayman's accelerator merging mechanism effectively saves 36% and 35% area for two area budgets. Cayman widely synthesizes reusable accelerators, each of which accelerates 3 distinct program regions on average from the benchmarks. The area saving percentage goes up to

TABLE II: Results under two area budgets 25% and 65%. #SB and #PR denote the number of sequential basic blocks and pipelined regions from selected kernels. #C, #D, and #S denote the number of *coupled*, *decoupled*, and *scratchpad* interfaces in synthesized accelerators. The "Area saving" columns denote the area-saving percentage by accelerator merging.

Suite	Benchmark	Small Area Budget (25%)								Large Area Budget (65%)								Runtime (s)
		Speedup(×)		Configurations					Area saving(%)	Speedup(×)		Configurations					Area saving(%)	
		over [21]	over [23]	#SB	#PR	#C	#D	#S		over [21]	over [23]	#SB	#PR	#C	#D	#S		
PolyB	3mm	35.4	14.0	16	9	0	0	15	74	117.9	46.8	16	9	0	0	15	70	69.2
	atax	16.5	6.5	9	9	0	9	1	38	32.2	12.7	9	9	0	0	10	21	15.5
	bicg	22.2	8.7	8	6	0	9	2	38	42.5	16.8	8	6	0	0	11	42	17.5
	doitgen	72.6	29.0	9	6	0	0	7	5	72.6	29.0	9	6	0	0	7	5	37.3
	mvt	13.2	5.2	8	9	1	8	0	26	19.5	7.7	8	9	0	0	9	21	18.9
	symm	13.6	9.2	4	3	3	6	1	38	37.8	25.7	4	3	0	0	10	35	1.4
	syrk	19.4	7.8	9	6	0	6	0	24	79.9	31.9	4	3	0	0	4	0	3.2
	trmm	19.0	7.6	4	3	2	4	0	38	19.0	7.6	4	3	2	4	0	38	0.0
	cholesky	17.0	15.7	12	6	3	8	0	26	17.0	15.7	12	6	3	8	0	26	0.1
	gramschmid..	20.3	8.3	11	12	1	14	0	24	91.6	37.7	5	6	0	0	9	48	47.2
	lu	17.5	16.3	16	9	3	12	0	26	17.5	16.3	16	9	3	12	0	26	0.2
	trisolv	15.3	6.0	8	6	1	5	6	38	15.3	6.0	8	6	1	5	6	38	0.1
	covariance..	18.2	7.3	4	3	5	4	0	38	18.2	7.3	4	3	5	4	0	38	0.1
	jacobi-2d	15.0	13.1	13	6	0	12	0	24	102.9	90.3	13	6	0	0	12	50	47.0
	deriche	5.6	4.5	26	18	0	20	0	34	13.5	10.1	28	21	0	21	0	39	0.1
	floyd-wars..	7.5	6.8	8	6	0	7	0	24	7.5	6.8	8	6	0	7	0	24	0.0
MachS	fft	42.6	38.5	5	14	4	22	2	26	42.6	38.5	5	14	4	22	2	26	0.2
	md	7.6	4.7	18	23	12	11	2	44	7.6	4.6	18	23	12	11	2	44	0.1
	spmv	8.6	8.6	11	11	4	5	0	44	9.5	9.5	9	14	5	5	0	26	0.2
	nw	10.8	4.7	5	24	0	9	1	38	10.8	4.7	5	24	0	9	1	38	0.1
Media	cjpeg	2.8	1.9	47	39	30	25	47	47	3.9	2.4	73	51	5	38	119	39	0.6
	epic	7.2	2.6	106	62	2	10	25	41	8.5	3.1	100	60	0	0	33	41	1068.1
CoreM	cjpeg-rose..	5.0	4.2	99	38	75	150	0	47	5.8	4.8	106	31	107	13	140	42	172.9
	zip-test	6.4	6.4	85	38	14	279	0	48	7.5	7.5	87	39	14	295	0	35	21.4
	parser-125..	5.0	4.3	3	4	0	0	5	43	5.0	4.3	3	4	0	0	5	43	100.4
	nnet-test	4.5	4.6	5	19	6	89	0	21	8.2	8.4	36	22	6	120	76	46	272.6
	linear-alg..	12.5	12.1	63	12	24	23	34	49	22.6	21.8	72	12	24	28	34	44	78.2
	loops-all-..	1.4	1.4	13	1	0	0	8	48	5.0	4.9	122	40	86	110	0	48	10.5
	average	14.4	8.0	22	14	7	27	6	36	27.2	15.0	28	16	10	25	18	35	70.8

74% and 70% for the 3mm benchmark, which includes 3 loops with identical basic blocks for merging. However, Cayman only saves 5% area for the doitgen benchmark since the benchmark only includes one hotspot region for acceleration and does not need accelerator merging. In practice, Cayman runs very fast, which only takes 70.8s on average.

Fig. 6 presents the speedup and area usage of Pareto-optimal solutions produced by the baselines and Cayman for four benchmarks from different suites. It shows that Cayman solutions achieve superior performance with different area usage for all the benchmarks. Specifically, NOVIA solutions are always located in the lower-left corner, indicating their limited speedup and low area overhead. Cayman also achieves comparable or better performance than NOVIA for low area budgets since Cayman explores a broader accelerator synthesis design space that fully covers NOVIA's. QsCores's performance speedup scales worse than Cayman when the area budget increases. Especially, QsCores cannot accelerate the loops-all-mid-10k-sp workload from CoreMark-Pro because it contains intensive control flow and memory access which cannot be accelerated by QsCores' sequential control and slow data access interface. Compared to full Cayman solutions, *coupled*-only ones achieve lower speedup for most benchmarks, demonstrating the significance of modeling and specializing processor-accelerator interfaces. The only exception is the loops-all-mid-10k-sp workload, where the difference between *coupled*-only and full Cayman solutions are minor. The reason is that loops in the workload commonly have loop-carried dependencies between floating-point operations, restricting the achievable pipeline II and circumventing the performance gains of adopting *decoupled* and *scratchpad* interfaces.

V. Conclusion

This paper proposes Cayman, an end-to-end framework for high-performance custom accelerator generation. Cayman models processor-accelerator data access interfaces and conducts efficient dynamic programming-based candidate selection, synthesizing accelerators with optimized control flow and data access. Cayman also introduces a novel accelerator merging mechanism to synthesize reusable accelerators. Experimental results on various benchmarks demonstrate that Cayman significantly outperforms the state-of-the-art frameworks.

Acknowledgment

This work was supported in part by the National Science Foundation of China (Grant No. T2325001) and Shanghai Municipal Science and Technology Major Project.

REFERENCES

[1] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–4.

[2] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, 2003, pp. 256–261.

[3] Bluespec. (2024) Bluespec accelerate-hls. [Online]. Available: https://info.bluespec.com/acceleratehls

[4] I. Brumar, G. Zacharopoulos, Y. Yao, S. Rama, D. Brooks, and G.-Y. Wei, "Early dse and automatic generation of coarse-grained merged accelerators," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 2, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3546070

[5] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 36. USA: IEEE Computer Society, 2003, p. 129.

[6] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, ser. FPGA '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 183–189. [Online]. Available: https://doi.org/10.1145/968280.968307

[7] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni, "An analysis of accelerator coupling in heterogeneous architectures," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015.

[8] M. Damian, J. Oppermann, C. Spang, and A. Koch, "Scaie-v: An open-source scalable interface for isa extensions for risc-v processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 169–174. [Online]. Available: https://doi.org/10.1145/3489517.3530432

[9] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin, *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.

[10] V. Gnanasambandapillai, J. Peddersen, R. Ragel, and S. Parameswaran, "Finder: Find efficient parallel instructions for asips to improve performance of large applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[11] X. Hao, H. Rong, M. Zhang, C. Sun, H. Jiang, and Y. Liang, "Popa: Expressing high and portable performance across spatial and vector architectures for tensor computations," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 199–210. [Online]. Available: https://doi.org/10.1145/3626202.3637566

[12] L. Jia, Z. Luo, L. Lu, and Y. Liang, "Tensorlib: A spatial accelerator generation framework for tensor algebra," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, 2021, p. 865–870. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586329

[13] L. Jia, Y. Wang, J. Leng, and Y. Liang, "Ems: efficient memory subsystem synthesis for spatial accelerators," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, ser. DAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 67–72. [Online]. Available: https://doi.org/10.1145/3489517.3530411

[14] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," ser. PLDI '94. New York, NY, USA: Association for Computing Machinery, 1994, p. 171–185. [Online]. Available: https://doi.org/10.1145/178243.178258

[15] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 127–136. [Online]. Available: https://doi.org/10.1145/3174243.3174264

[16] C. Li, Y. Wang, H. Li, and Y. Han, "Append: Rethinking asip synthesis in the era of ai," ser. DAC '23, 07 2023, pp. 1–6.

[17] G. Liu, J. Primmer, and Z. Zhang, "Rapid generation of high-quality risc-v processors from functional instruction set specifications," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[18] J. Oppermann, B. M. Damian-Kosterhon, F. Meisel, T. Mürmann, E. Jentzsch, and A. Koch, "Longnail: High-level synthesis of portable custom instruction set extensions for risc-v processors from descriptions in the open-source coredsl language," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24, 2024. [Online]. Available: https://doi.org/10.1145/3620666.3651375

[19] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, 2006.

[20] A. Sohrabizadeh, C. H. Yu, M. Gao, and J. Cong, "Autodse: Enabling software programmers design efficient fpga accelerators," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 147. [Online]. Available: https://doi.org/10.1145/3431920.3439464

[21] D. Trilla, J.-D. Wellman, A. Buyuktosunoglu, and P. Bose, "Novia: A framework for discovering non-conventional inline accelerators," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 507–521. [Online]. Available: https://doi.org/10.1145/3466752.3480094

[22] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 205–218, 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1735970.1736044

[23] G. Venkatesh, J. Sampson, N. Goulding-Hotta, S. K. Venkata, M. B. Taylor, and S. Swanson, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[24] Y. Xiao, Z. Luo, K. Zhou, and Y. Liang, "Cement: Streamlining fpga hardware design with cycle-deterministic ehdl and synthesis," in *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 211–222. [Online]. Available: https://doi.org/10.1145/3626202.3637561

[25] Xilinx Inc. (2024) Vitis high-level synthesis user guide (ug1399). [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls

[26] R. Xu, Y. Xiao, J. Luo, and Y. Liang, "Hector: A multi-level intermediate representation for hardware synthesis methodologies," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3508352.3549370

[27] H. Ye, C. Hao, J. Cheng, H. Jeong, J. Huang, S. Neuendorffer, and D. Chen, "ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation," 2021. [Online]. Available: http://arxiv.org/abs/2107.11673

[28] C. H. Yu, P. Wei, M. Grossman, P. Zhang, V. Sarker, and J. Cong, "S2fa: an accelerator automation framework for heterogeneous computing in datacenters," ser. DAC '18, New York, NY, USA, 2018. [Online]. Available: https://doi.org/10.1145/3195970.3196109

[29] G. Zacharopoulos, A. Ejjeh, Y. Jing, E.-Y. Yang, T. Jia, I. Brumar, J. Intan, M. Huzaifa, S. Adve, V. Adve, G.-Y. Wei, and D. Brooks, "Trireme: Exploration of hierarchical multi-level parallelism for hardware acceleration," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 3, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3580394

[30] G. Zacharopoulos, L. Ferretti, G. Ansaloni, G. Di Guglielmo, L. Carloni, and L. Pozzi, "Compiler-assisted selection of hardware acceleration candidates from application source code," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 129–137.

[31] G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, and L. Pozzi, "Regionseeker: Automatically identifying and selecting accelerators from application source code," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[32] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.