Data-Dependent Loop Approximations for Performance-Quality Driven High-Level Synthesis

Seogoo Lee and Andreas Gerstlauer
Department of Electrical and Computer Engineering, The University of Texas at Austin
{sglee, gerstl}@utexas.edu

***Abstract*—Approximate computing (AC) obtains performance or energy gains by trading off computational accuracy in error-tolerant applications. At the hardware level, AC-aware high-level synthesis (HLS) tools apply operation-level approximations to generate a quality-reduced register-transfer level design from an accurate high-level description. While existing tools primarily target energy savings, we focus on performance optimizations. Loops are often the most performance-critical application code structures. In a loop, iterations can have different impacts on output quality due to an inherent data-dependency of approximations. Exploiting iteration-wise data variations, we present a novel technique that clusters iterations according to data statistics and applies different approximations in each cluster to maximize performance gains. Performance can be improved by up to 76%, with up to 21.7% stemming from clustering.**

I. Introduction

High-level synthesis (HLS) tools are widely used in designing hardware accelerators for compute-intensive applications. They automatically generate a register-transfer level (RTL) design from a high-level description. For inherently error-tolerant applications, recent work [1], [2] incorporates approximating computing (AC) concepts into HLS by applying operation-level approximations to trade off computational quality. Existing AC-aware HLS tools target energy savings. In this paper, we study a novel approach that targets performance gains.

Loops are often the performance-critical parts of applications, and as such have been studied extensively in traditional HLS. Unrolling and pipelining are two widely used optimization techniques. A wide range of advanced loop optimizations have been proposed until recently [3], [4], [5], but none of them considers quality as a design metric. The quality impact of hardware approximations is inherently data-dependent. At the same time, data statistics can vary across loop iterations. Existing AC-aware HLS tools fully unroll all loops or treat all iterations the same. Complete unrolling of loops allows fine-grain, data-specific optimizations to be applied at the individual operation level, but breaks the regularity of loop structures and results in high area and control overheads, especially for large iteration counts. By contrast, keeping loops rolled and approximating all iterations in the same way ignores iteration-wise variations, which is suboptimal.

We propose an approach that enables fine-grain, iteration-specific optimizations, while keeping overall loop structures intact. We cluster loop iterations according to their similarity in data statistics and apply different approximations for each cluster. In doing so, we employ a quality-performance optimization approach that automatically finds the best iteration clusters and their approximation levels. Our goal is performance maximization under quality constraints, and, without loss of generality, we utilize operation eliminations as hardware approximation technique. This approximation removes operations that have a small impact on output quality by replacing their output with zero [6], [1]. Eliminated operations can be effectively exploited during HLS scheduling to reduce clock cycles and hence increase performance.

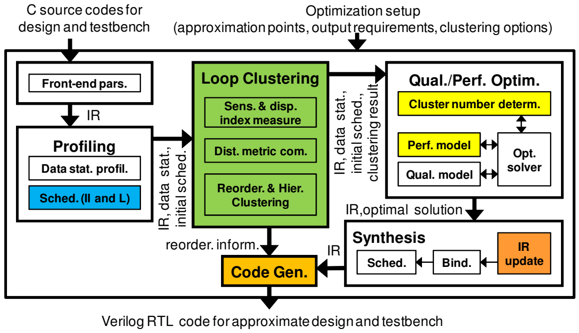

C source codes for design and testbench
Optimization setup (approximation points, output requirements, clustering options)
Front-end pars.
IR
Profiling
Data stat. profil.
Sched. (II and L)
IR, data stat., initial sched.
Loop Clustering
Sens. & disp. index measure
Dist. metric com.
Reorder. & Hier. Clustering
IR, data stat., initial sched., clustering result
Qual./Perf. Optim.
Cluster number determ.
Perf. model
Qual. model
Opt. solver
IR,optimal solution
Synthesis
Sched.
Bind.
IR update
reorder. inform.
IR
Code Gen.
Verilog RTL code for approximate design and testbench

Fig. 1: Overview of our loop optimization framework.

II. Overview

We integrate our loop optimization into the AC-aware HLS tool from [1], which is built on top of the LLVM-based LegUp HLS framework [7]. Fig. 1 shows an overview of our optimization flow with differences from [1] highlighted. The work in [1] automatically generates approximated RTL designs from an accurate high-level C description under given output quality constraints, and an optimization setup that includes user-specified or automatically identified approximation points, which are variables in the high-level C description to apply approximations to. In [1], a profiling step collects data statistics from simulations of the accurate design and performs a pre-scheduling to obtain operation mobilities and total latency (L). In our case, we also obtain the initiation interval (II) of all loops specified to be pipelined by the user.

The main extension in our flow is the loop clustering step inserted after profiling. We use iteration-wise data statistics at approximation points weighted by their estimated quality and noise sensitivity to compute a distance metric used for clustering. A hierarchical clustering algorithm then determines the optimal mappings of iterations to clusters, for all possible numbers of clusters from one to the number of iterations. In the process, we reorder iterations (if possible) to find optimal clusters and maximize gains.

Clustering results are given to the final optimization and code generation steps. The optimization finds a solution that maximizes approximation benefits. We build on the solver from [1], which uses an efficient heuristic search coupled with semi-analytical quality and energy models to find quality-energy optimized designs. We replace the energy model with a performance model and iteratively run the solver to find the best number of clusters and their approximations that maximize performance gains. With the optimization results, we modify the rolled dataflow graph (DFG) and execute HLS scheduling and binding to generate the Verilog RTL code.

III. Selective Loop Optimization

Fig. 2 illustrates our loop optimization concept. Let the accurate high-level description of the input design contains a loop with I iterations and P approximation points that are mapped to V initial decision variables ($P = V$). Note that decision variables are defined only within the loop body and are not indexed by loop iterations. We use profiled signal powers $D_{p,i}$ of approximation points $p \in \{0, \cdots, P-1\}$ in iterations $i \in \{0, \cdots, I-1\}$ for clustering. After the optimization, the loop is split into C clusters that have different approximations and iteration counts, s.t. the average quality across all iterations meets constraints and the total sum of iteration counts is I.

A. Profiling

Using the given testbench, we run a C simulation of the unrolled design to obtain statistical information $D_{p,i} = \mu_{p,i}^2 + \sigma_{p,i}^2$ of the application data at each approximation point p in each iteration i, where $\mu_{p,i}$ is the mean and $\sigma_{p,i}^2$ the variance. Profiled statistics are used for clustering and quality estimation, i.e. final synthesis results depend on inputs used in profiling.

We also run the scheduler from [7] on the rolled loops in the input design to extract the accurate loop body's latency (L_{ac}) and, if pipelined, initiation interval (II_{ac}). We also obtain the scheduling table $\mathbf{MAP}_{\text{sch}}$, where $\mathbf{MAP}_{\text{sch}}(t)$ returns the set of operations scheduled in clock cycle t.

Finally, we build a mapping table $\mathbf{MAP}_{\text{as}}$ of all operations in the loop body that can be jointly eliminated. For example, if an addition is eliminated, i.e. approximated to zero, all of its predecessor operations become unnecessary and can also be eliminated. In addition, any adder successors can be bypassed, and multiplier successors can be eliminated. $\mathbf{MAP}_{\text{as}}$ is used for performance estimation during our optimization step.

B. Loop Clustering

Using the profiling and scheduling information, we perform loop clustering according to iteration-wise data statistics.

1) Distance Metric: We cluster iterations using a distance metric computed from the P-dimensional vectors $\mathbf{D}_i = (D_{0,i}, \cdots, D_{P-1,i}), i \in \{0, \cdots, I-1\}$. We first scale data statistics $\mathbf{D}_i$ to account for their quality and noise sensitivity.

Some elements in $\mathbf{D}_i$ should contribute less to clustering due to their small impact on output quality. For example, an approximation at a multiplier input causes a larger noise than the same approximation at a multiplier output. Therefore, when we cluster iterations, iteration-wise differences in multiplier outputs should be weighted down. This quality impact is a function of the DFG and data statistics. For each approximation point p, we analytically estimate its impact as the output noise δ_p resulting from a truncation of 1 least significant bit using the method in [8]. We then use the δ_p normalized against the maximum $\delta_{\max}$ to compute a weighted $\mathbf{D}'_i = (D'_{0,i}, \cdots, D'_{P-1,i}), D'_{p,i} = D_{p,i} \cdot \delta_p / \delta_{\max}$.

In addition, some elements in $\mathbf{D}_i$ may have larger variations across iterations but a smaller average value than others. Due to the relative nature of noise, such elements should be counted more in determining clusters. To apply such effects, we use dispersion indices $\phi_p = \sigma_p^2 / \mu_p$ computed as the ratio of variance σ_p^2 and mean μ_p of iteration-wise signal powers $\{D_{p,0}, D_{p,1}, \cdots, D_{p,I-1}\}$ at approximation point p. We normalize ϕ_p against their maximum $\phi_{\max}$ to compute a scaled $\mathbf{D}''_i = (D''_{0,i}, \cdots, D''_{P-1,i}), D''_{p,i} = D'_{p,i} \cdot \phi_p / \phi_{\max}$.

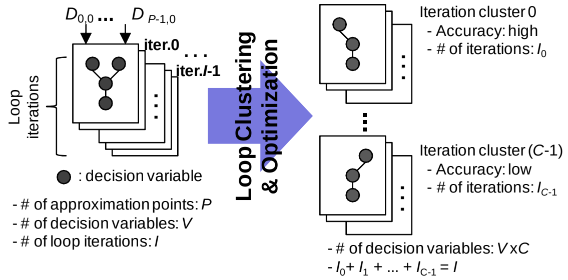

$D_{0,0}$ ... $D_{P\text{-}1,0}$
iter.0
...
iter.I-1
Loop iterations
● : decision variable
- # of approximation points: P
- # of decision variables: V
- # of loop iterations: I
Loop Clustering & Optimization
Iteration cluster 0
- Accuracy: high
- # of iterations: I_0
Iteration cluster (C-1)
- Accuracy: low
- # of iterations: $I_{C\text{-}1}$
- # of decision variables: VxC
- $I_0 + I_1 + ... + I_{C\text{-}1} = I$

Fig. 2: Iteration clustering and optimization.

The $P \times I$ multi-dimensional vector $\mathbf{D}''$ becomes the input to clustering. We apply two commonly used distance metrics: Euclidean and cosine distances. The Euclidean distance between iteration i and j is calculated as $M_{i,j}^{\text{EU}} = \sqrt{\sum_{p \in \text{P}} (D''_{i,p} - D''_{j,p})^2}$. This is simple, but does not differentiate elements that contribute to a distance. A cosine distance $M_{i,j}^{\text{CS}}$ is instead known to give better results in multi-dimensional clustering.

$$M_{i,j}^{\text{CS}} = 1 - \frac{\mathbf{D}''_i \cdot \mathbf{D}''_j}{|\mathbf{D}''_i||\mathbf{D}''_j|} = 1 - \frac{\sum_{p \in \text{P}} D''_{i,p} \times D''_{j,p}}{\sqrt{\sum_{p \in \text{P}} D''^2_{i,p}} \sqrt{\sum_{p \in \text{P}} D''^2_{j,p}}}. \quad (1)$$

We build an $I \times I$ matrix $\mathbf{M}$ out of elements $M_{i,j}$ representing the distances of iterations i and j. $\mathbf{M}$ is symmetric with zero diagonal elements.

2) Iteration Reordering and Clustering: In many cases, data statistics $\mathbf{D}_i$ are not aligned with iteration sequences, making clustering along existing boundaries non-optimal. Better clusters can be found if we reorder iterations according to $M_{i,j}$. Reordering is applicable for loops that do not have loop-carried dependencies. Even if there exists a dependency, loops that successively compute a sum of all iteration outputs, such as filters, can be reordered.

Our reordering algorithm first finds an initial iteration pair (i and j) that has a minimum distance. This pair becomes the seed for reordering. We then find the iteration that is closest to i or j, and append this iteration before i or after j to become the new first or last iteration, respectively. We successively append iterations until there is none remaining. The result is a mapping table $\mathbf{MAP}_{\text{ro}}(i')$ that returns the original iteration index i for each reordered iteration i'. We also collect distances between adjacent reordered iterations, $M'_{i',i'+1}, \forall i' \in \{0, \cdots, I-2\}$. We pass this information to clustering. If reordering is not applied, we directly pass the original iteration indices and adjacent distances $M_{i,i+1}, \forall i \in \{0, \cdots, I-2\}$ to clustering.

K-Means and hierarchical clustering are widely-used clustering algorithms. We adopt hierarchical clustering since it provides results for different numbers of clusters and does not have a random behavior in its initial seed selection. Hierarchical clustering is performed using the mappings and distances from reordering. It iteratively merges adjacent iterations that have minimum distance into clusters. We run hierarchical clustering starting from I down to 1 cluster, and record the intermediate results for all possible cluster counts $C = \{1, 2, \cdots, I\}$. For each C, the result is a mapping table $\mathbf{MAP}_C$ of iterations to clusters. $\mathbf{MAP}_C(c)$ returns the set of iteration indices in the c-th of C clusters ($0 \leq c < C$).

C. Quality-Performance Optimization

Our optimization problem is to find the best number of clusters and their approximations that maximize performance benefits under quality constraints. Our decision variables $s_{p,c}$ are binary, where $s_{p,c} = 1$ iff the operation at approximation point p in cluster c is eliminated, and $s_{p,c} = 0$, otherwise. We propose a heuristic that iteratively increments the number of clusters and finds approximations for all clusters until no more performance gain is achieved. Our heuristic reuses the optimization solver from [1]. For a given clustering and corresponding decision variables, the solver in [1] can find near-optimal approximations with a dramatically reduced complexity using analytical quality and energy models. We replace the energy model in [1] with a performance model. For quality estimation, we use the analytical quality model from [1].

1) Performance Model: Processing time is the product of the number of clock cycles T and the maximum critical path delay across all cycles. The latter can be reduced by other types of approximations, such as precision scaling, but only when the approximation is applied in the most critical cycles across the entire design, which limits benefits. Therefore, in this paper, we only consider operation eliminations targeting reductions of T for performance improvement.

T is determined from the II and L of each cluster. We estimate II_c and L_c of cluster c using $\mathbf{MAP}_{\text{sch}}$ and $\mathbf{MAP}_{\text{as}}$ of the accurate design obtained during profiling. When approximating operation p in cluster c, it can open a set of other operation eliminations. By excluding operations in $\mathbf{MAP}_{\text{as}}(p)$ for all eliminated p in cluster c from $\mathbf{MAP}_{\text{sch}}$, we determine the number of clock cycles that do not have any operations scheduled after approximations, which results in an estimated latency reduction of L_c^{red}. The new latency of the cluster thus becomes $L_c = L_{\text{ac}} - L_c^{\text{red}}$.

If pipelining is applied, the II of each cluster is estimated by similarly checking how many memory accesses ($N_{\text{mem}}^{\text{red}}$) and arithmetic operations ($N_{\text{add}}^{\text{red}}$ and $N_{\text{mul}}^{\text{red}}$) using shared resources can be eliminated by approximations. For cluster c, the remaining numbers of operations are computed as $N_{op,c} = N_{op,\text{ac}} - N_{op,c}^{\text{red}}$, where $op \in \{\text{mem}, \text{add}, \text{mul}\}$ is the operation type. Here, $N_{op,\text{ac}}$ are the number of respective operations in the accurate loop body.

With $N_{op,c}$, we follow the initial II determination method under resource constraints in [7] and estimate a new II_c as $II_c = \min_{op}(N_{op,c}/N_{op}^{\text{con}})$. Here, $N_{\text{mem}}^{\text{con}}$ is the constraint on the number of memory ports, e.g. $N_{\text{mem}}^{\text{con}} = 2$ for a dual port memory. N_{op}^{con} is the resource constraint for arithmetic operation type $op \in \{\text{add}, \text{mul}\}$.

With estimated L_c and II_c, the total number of clock cycles for a pipelined design is estimated as $T = \sum_{c \in C} II_c \times (I_c - 1) + L_c$. Here, I_c is the iteration count of cluster c. The II becomes the main target for reductions through approximations. Without pipelining, $T = \sum_{c \in C} L_c \times I_c$, and only a reduction in L_c matters.

2) Optimization: Our heuristic iteratively calls the solver in [1] to obtain an approximation that maximizes a reduction in T for a given number of clusters C, successively increasing C starting from $C = 1$, i.e. without clustering, up to C_{MAX}. For a given C, we first update decision variables to be used in the solver, using the clustering result $\mathbf{MAP}_C$. Before clustering, V decision variables and their mappings to approximation points are identified in the accurate, unclustered loop body. For a C larger than 1, each cluster has its own V decision variables, and the V original decision variables are expanded into $V \times C$ new decision variables according to $\mathbf{MAP}_C$.

TABLE I: Example setup.

Design	V	I	Dataset	Description	T_{ac}	Pipelined
conv2d	2	25	stitch [9]	5×5 2D conv. filter	226	No
quatmul	4	80	localization [9]	quat product	731	Yes
idct	8	64	JPEG (lena)	1D inv. DCT	1107	Yes

The solver from [1] uses a breadth-first search to successively evaluate different approximation solutions. For each solution, we modify the solver to first find its effective number of clusters, C_{eff}. C_{eff} is the number of clusters that truly have different approximations. For example, if two among $C = 3$ clusters have the same approximations, there is no need to leave the three clusters separated. Instead, we merge the two clusters into one, and C_{eff} becomes 2. We then let the solver estimate the quality and performance for the effective clusters and their approximations.

The solver returns the evaluated solution that has the smallest T. We stop increasing C if T does not improve any more. The last solution consisting of the best $C_{\text{eff}}^{\text{opt}}$ and corresponding approximations is returned as the final optimization result.

D. Synthesis

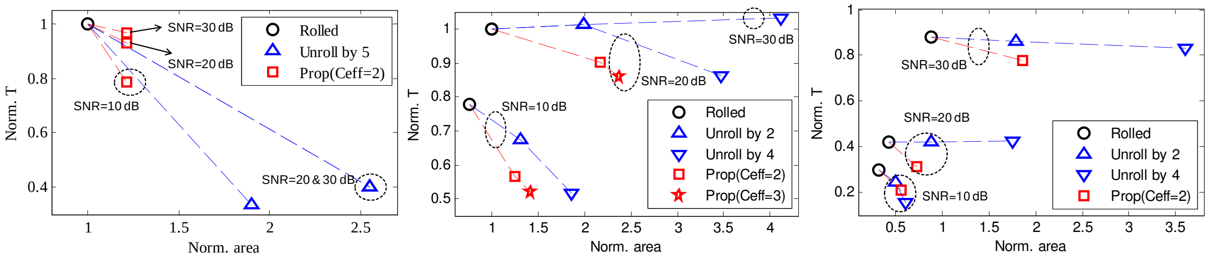
With the final solution from the optimization, we synthesize approximated Verilog RTL code. We first modify the intermediate representation (IR) of the accurate design to create multiple serial copies of a loop. Each effective cluster's approximation level and the number of iterations in each cluster is then used to adjust the code for each loop. After modifying the IR, we run standard scheduling and binding in HLS. When generating a RTL code, $\mathbf{MAP}_{\text{ro}}$ is applied for iteration reordering as an address lookup table.

IV. Results

We present experimental results for three examples from [1]. Table I summarizes our setup. All examples are constrained by their output signal-to-noise ratio (SNR). In all cases, all external inputs and internal multiplier outputs were manually selected as approximation points. *conv2d* is not pipelined, while all other examples are. For all experiments, we allowed the tool to explore up to 5 clusters ($C_{\text{MAX}} = 5$). For the performance model, $N_{\text{mem}}^{\text{con}} = 1$ and $N_{\text{mul}}^{\text{con}} = 4$ are assumed. There is no resource constraint on the number of adders.

Table II summarizes results. For the three examples, we run our clustering optimization for a relatively low (10 dB), medium (20 dB), and high (30 dB) SNR, and we compare performance from our approach to that of an accurate design (T_{ac}). We also compare our performance against an approximated design without clustering (NC). For each SNR, we apply 3 optimization levels; (1) clustering with raw data statistics and no iteration reordering (C); (2) clustering using the modified data statistics, but no iteration reordering (C+D$''$); and (3) clustering using all optimizations (C+D$''$+RO). Euclidean distance is used as the default metric. We also provide results using cosine distances for C+D$''$+RO+Cos.

Gains vary aross examples and SNR levels. Maximum performance improvements of 69% and 76% are observed in *idct* at 20 dB and 10 db SNRs. This corresponds to a PSNR



(a) *conv2d*. (b) *quatmul*. (c) *idct*.

Fig. 3: Performance-area design space.

TABLE II: Optimization results compared to T_{ac} in Table I.

Design	SNR	T [cycl.] / $C_{\mathrm{eff}}^{\mathrm{opt}}$				
	[dB]	NC	C	C+D$''$	C+D$''$+RO	C+D$''$+RO+Cos.
conv2d	30	226 (0%)	226 / 1	226 / 1	226 / 1 (0%)	218 / 2 (-3.5%)
	20	226 (0%)	224 / 2	226 / 1	226 / 1 (0%)	210 / 2 (-7%)
	10	226 (0%)	222 / 2	212 / 2	178 / 2 (-21.2%)	178 / 2 (-21.2%)
quatmul	30	731 (0%)	731 / 1	731 / 1	731 / 1 (-0%)	731 / 1 (-0%)
	20	731 (0%)	731 / 1	731 / 1	651 / 3 (-10.9%)	657 / 2 (-10.1%)
	10	573 (-21.6%)	573 / 1	573 / 1	414 / 3 (-43.3%)	414 / 3 (-43.3%)
idct	30	981 (-11.4%)	981 / 1	981 / 1	891 / 2 (-19.5%)	915 / 2 (-17.3%)
	20	477 (-56.9%)	477 / 1	477 / 1	348 / 2 (-68.6%)	477 / 1 (-56.9%)
	10	351 (-68.3%)	351 / 1	351 / 1	267 / 2 (-75.9%)	315 / 2 (-71.5%)

of 32 dB and 22 dB, where the accurate *idct* has a PSNR of 48 dB. The maximum gain from clustering is 21.7% in *quatmul* at the lowest SNR. The *idct* has the largest iteration-wise variation in its input data, but NC baseline gains limit additional clustering benefits (up to 11.7%). Other examples have smaller iteration-wise variations, but little to no NC gains, and thus larger clustering gains. The choice between Euclidean and cosine distances depends on the application and inputs. In *conv2d*, cosine distances provide higher or the same gains than Euclidean ones for all SNRs. By contrast, Euclidean distances generally work better in other examples. In general, C+D$''$+RO gives the best results. There is no clustering gain in *quatmul* at 30 dB since there is no opportunity for operation eliminations, and having multiple clusters only introduces overhead.

Even though we evaluate up to 5 clusters, $C_{\mathrm{eff}}^{\mathrm{opt}}$ ends up being less than 4 in all cases. For pipelined examples, this is due to pipelining overhead. Having multiple clusters only provides benefits if the number of reduced clock cycles due to II reduction can compensate for the overhead in breaking the pipelining structure. For the non-pipelined *conv2d*, $C_{\mathrm{eff}}^{\mathrm{opt}}$ is still limited to 2. This is because the example is simple, having only one multiplication and addition in its loop body. As such, there exist only two possible approximation levels for the loop body, and C_{eff} can not be larger than 2.

Due to control and resource replication in our HLS tool, creating multiple clusters increases the hardware area. This motivates us to compare our proposed approach to existing loop unrolling techniques, which provide similar area versus performance tradeoffs. Unrolling can open better scheduling opportunities while also allowing for approximations at finer granularity. Note that, clustering is orthogonal and can be combined with existing loop optimization techinques, such as unrolling. Fig. 3 shows the quality, area, and performance design space for *conv2d* (Fig. 3a), *quatmul* (Fig. 3b), and *idct* (Fig. 3c) examples using either unrolling or clustering with different unroll factors and effective cluster counts, respectively. The x-axes show area results from Synopsys DesignCompiler, normalized against the area of the accurate rolled design. The y-axes show the T of approximated designs normalized against T_{ac}. For *conv2d*, due to its simplicity, there are no approximation opportunities without unrolling and clustering at any SNR level. Furthermore, unrolling can only be done by factors of 5, which is the smallest divisor of the loop iteration count. By contrast, clustering has no such restrictions. At 30 dB in *quatmul* and at 20 and 30 dB in *idct*, due to their effective pipelining, there are no scheduling benefits from unrolling. Only when additional approximations are enabled at lower SNRs, benefits are seen. Overall, as results show, clustering can either dominate unrolled designs or provide new Pareto choices. At all SNRs in *conv2d*, at 20 dB in *quatmul*, and at 10 and 30 dB in *idct*, designs with multiple clusters work as new Pareto-optimal design points next to the partially unrolled designs. For *quatmul* at 10 dB and *idct* at 20 dB, the designs with 2 clusters even dominate unrolled designs.

Runtime of our tool is dominated by the solver from [1], which depends on the size of the design space to explore. On an Intel Core i7 at 2.67GHz, the average runtime per SNR target with vs. without clustering is 8s vs. 3s for *conv2d*, 9 min. vs. 3 min. for *quatmul*, and 25 min. vs. 8 min. for *idct*.

V. Summary and Conclusions

In this paper, we presented our loop optimization technique for quality-performance-area aware AC-HLS. We proposed an approach that uses operation eliminations for performance optimization and selectively approximates loop iterations according to their data-dependent impact on final output quality. Our approach uses an effective optimization heuristic to automatically determine the optimal number of clusters and approximations including loop transformations. Result show that up to 76% of performance improvements can be achieved from approximations, where up to 22% come from applying finer-grained approximations due to clustering. We currently only support loops in which iterations can be reordered. In the future, we plan to extend our approach to dependent and nested loop structures as well as other approximation techniques.

References

[1] S. Lee and A. Gerstlauer, “High-level synthesis of approximate hardware under joint precision and voltage scaling,” in *DATE*, March 2017.

[2] C. Li *et al.*, “Joint precision optimization and high level synthesis for approximate computing,” in *DAC*, June 2015.

[3] M. Tan *et al.*, “Elasticflow: A complexity-effective approach for pipelining irregular loop nests,” in *ICCAD*, November 2015.

[4] J. Liu *et al.*, “Loop splitting for efficient pipelining in high-level synthesis,” in *FCCM*, May 2016.

[5] W. Zuo *et al.*, “Improving polyhedral code generation for high-level synthesis,” in *CODES+ISSS*, September 2013.

[6] K. Nepal *et al.*, “ABACUS: A technique for automated behavioral synthesis of approximate computing circuits,” in *DATE*, March 2014.

[7] A. Canis *et al.*, “LegUp: High-level synthesis for FPGA-based processor/accelerator systems,” in *FPGA*, February 2011.

[8] S. Lee *et al.*, “Statistical modeling of approximate hardware,” in *ISQED*, March 2016.

[9] S. K. o. Venkata, “SD-VBS: The San Diego vision benchmark suite,” in *IISWC*, 2009.