Supplementary Material for "Towards a Comprehensive Benchmark for FPGA Targeted High-Level Synthesis"

Anonymous Author(s)

A Dataset

URL to website/platform where the dataset/benchmark can be viewed and downloaded: `https://zenodo.org/record/8034115`.

The license of our repository is Creative Commons Attribution 4.0 International. Our benchmark is for researchers and scientists to propose innovative methods for HLS. We implement a number of representative methods and compare all of them under the same framework.

We plan to maintain the dataset and code base by creating a GitHub repository upon paper acceptance. The kernels are chosen from the commonly-used benchmarks: MachSuite [2] and PolyBench [3]. We will continuously add new kernels and labeled designs with instructions on how to download and use them in the upcoming GitHub repository. We plan to add all the kernels, along with their varying input sizes, from the aforementioned benchmarks, in addition to expanding to new benchmarks.

B Reproducibility

To ensure all the results are easily reproducible, we release the complete set of hyperaprameters used to train the methods. In the main paper, we have 10 methods in total, where each method is trained on both versions of the dataset (SDx (v1) and Vitis (v2)) as well as both the regress. Therefore, there are $10 * 2 * 2 = 40$ models in total to be trained. Each one of these 40 experiments has its complete set of hyperparameters released under the folder "`configs_for_reproducibility`". Specifically, the files under the folder "`configs_for_reproducibility`" are of the following formats: "`<method_name>_v<1|2>_<r|c>.txt`". For example, the file "`[CODET5,GNN-GSE]_v1_c.txt`" indicates this file contains all the hyperparameters we used to train the concatenation-based model on SDx (v1) for the classification task. As a result, we encourage referring to these files for complete hyperaprameter details, such as learning rate, batch size, number of GNN/transformer layers, etc.

C Pragma Effects

Example 1: pragma TILE

Code 1 illustrates the `atax` kernel from PolyBench [3], which includes two matrix-vector multiplications. The objective of this example is to demonstrate the influence of the TILE pragma. Consequently, only the TILE pragma for the i loop is modified from its default value to 4 (Line 41), while the remaining pragmas retain their default settings, indicating they are turned off. The TILE pragma can be used to adjust the memory footprint of the loop below it. Specifically, with a TILE factor of 4 in this example, we aim to cache data for 4 iterations of the i loop. Without this pragma, the Merlin Compiler [1] sets a default caching amount.

Submitted to the 37th Conference on Neural Information Processing Systems (NeurIPS 2023) Track on Datasets and Benchmarks. Do not distribute.

```
#pragma ACCEL kernel
void kernel_atax(double A[116][124], double x[124], double y[124], double tmp[116]){
  int i, j;

  for (i = 0; i < 124; i++)
    y[i] = ((double )0);

#pragma ACCEL PIPELINE off
#pragma ACCEL TILE FACTOR=4
#pragma ACCEL PARALLEL FACTOR=1
  for (i = 0; i < 116; i++) {
    tmp[i] = 0.0;

#pragma ACCEL PARALLEL reduction=tmp FACTOR=1
    for (j = 0; j < 124; j++) {
      tmp[i] += A[i][j] * x[j];
    }

#pragma ACCEL PARALLEL reduction=y FACTOR=1
    for (j = 0; j < 124; j++) {
      y[j] += A[i][j] * tmp[i];
    }
  }
}
```

Code 1: Code snippet of the atax kernel with a TILE pragma as input to the Merlin Compiler.

```
// skipping the definition of helper memcpy modules for brevity
__kernel void kernel_atax(merlin_uint_256 A[3596],double x[124],double y[124],double tmp[116]){
#pragma HLS INTERFACE m_axi port=A offset=slave depth=3596 bundle=merlin_gmem_kernel_atax_256_0
#pragma HLS INTERFACE m_axi port=tmp offset=slave depth=116 bundle=merlin_gmem_kernel_atax_64_tmp
#pragma HLS INTERFACE m_axi port=x offset=slave depth=124 bundle=merlin_gmem_kernel_atax_64_0
#pragma HLS INTERFACE m_axi port=y offset=slave depth=124 bundle=merlin_gmem_kernel_atax_64_y

#pragma HLS INTERFACE s_axilite port=A bundle=control
#pragma HLS INTERFACE s_axilite port=tmp bundle=control
#pragma HLS INTERFACE s_axilite port=x bundle=control
#pragma HLS INTERFACE s_axilite port=y bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  double y_buf[124];
  int i;
  int j;

  merlinL10: for (i = 0; i < 124; i++) {
#pragma HLS dependence variable=y_buf array inter false
#pragma HLS pipeline
    y_buf[i] = ((double )0);
  }
  merlin_memcpy_0(y,0,y_buf,0,sizeof(double ) * ((unsigned long )124),992UL);

  merlinL9: for (i = 0; i < 29; i++) {
    double y_buf_0[124];
    double x_buf[124];
    double A_buf[4][124];
#pragma HLS array_partition variable=A_buf cyclic factor=4 dim=2
    double tmp_buf[4];
    merlin_memcpy_1(tmp_buf,0,tmp,i * 4,sizeof(double ) * ((unsigned long )4),32UL);
    memcpy_wide_bus_read_double_2d_124_256(A_buf,(size_t )0,(size_t )0,(merlin_uint_256 *)A,(size_t
        )(((long )i) * 496L * ((long )8)),sizeof(double ) * ((unsigned long )496L),(size_t )496L);
    merlin_memcpy_2(x_buf,0,x,0,sizeof(double ) * ((unsigned long )124),992UL);
    merlin_memcpy_3(y_buf_0,0,y,0,sizeof(double ) * ((unsigned long )124),992UL);

    merlinL8: for (int i_sub = 0; i_sub < 4; ++i_sub) {
      tmp_buf[i_sub] = 0.0;
      merlinL7: for (j = 0; j < 124; j++) {
#pragma HLS pipeline
        tmp_buf[i_sub] += A_buf[i_sub][j] * x_buf[j];
      }
      merlinL6: for (j = 0; j < 124; j++) {
#pragma HLS dependence variable=y_buf_0 array inter false
#pragma HLS pipeline
        y_buf_0[j] += A_buf[i_sub][j] * tmp_buf[i_sub];
    } }
    merlin_memcpy_4(y,0,y_buf_0,0,sizeof(double ) * ((unsigned long )124),992UL);
    merlin_memcpy_5(tmp,i * 4,tmp_buf,0,sizeof(double ) * ((unsigned long )4),32UL);
} }
```

Code 2: AMD/Xilinx Vitis HLS equivalent of Code 1 automatically generated by the Merlin Compiler.

The Merlin Compiler takes Code 1 as input and applies source-to-source code transformations to generate an equivalent implementation using AMD/Xilinx Vitis HLS. The resulting code is shown in Code 2. To keep it concise, we omit the helper modules (e.g., `memcpy` modules) responsible for transferring data from off-chip (global) memory to on-chip buffers. The HLS code is then processed by the AMD/Xilinx HLS tool, which generates the RTL-equivalent of the design that can be mapped to the target FPGA. We do not display the generated RTL code here as it exceeds 10,000 lines.

The generated HLS code begins with `interface` pragmas, which define the communication protocol between the design and external components. Notably, in the top-level function, the precision of the argument `A` is changed from double (64 bits) to 256-bit data. This technique, known as memory coalescing, helps reduce communication latency by combining multiple (in this case, 4) memory accesses into a single transaction. The Merlin Compiler automatically determines the appropriate approach for memory coalescing. Additionally, it analyzes data dependencies within the design, assisting the HLS tool in understanding whether true dependencies exist between iterations of different loops (lines 77 and 102). The Merlin Compiler can also infer other AMD/Xilinx HLS pragmas, such as `pipeline`, `array_partition`, `unroll`, `inline`, etc.

Since the second i loop in Code 1 is tiled with a factor of 4, it is transformed into two sub-loops in Code 2, namely lines 83 (merlinL9 loop with trip-count of 116/4=29) and 95 (merlinL8 loop with trip-count of 4). The merlinL9 loop transfers the necessary data for executing the merlinL8 loop before proceeding with its execution.

Example 2: pragma PIPELINE and PARALLEL

Code 3 showcases the `mvt` kernel from PolyBench, which, similar to `atax`, involves two matrix-vector multiplications. However, the order of execution and the dependency between them differ. In the first nested loop, we apply a PIPELINE pragma with the *flatten* mode, resulting in unrolling the inner loops and pipelining the loop at which the pragma is applied. This technique is known as fine-grained pipelining. To highlight the contrast with coarse-grained pipelining, we employ a different form of pipelining for the second nested loop. Coarse-grained pipelining divides the computation within the loop into multiple modules (e.g., load-compute-store units) connected in a pipeline fashion. Additionally, we apply PARALLEL pragmas to the second nested loop. The remaining pragmas, set to their default values, have no impact on the resulting microarchitecture.

```
#pragma ACCEL kernel
void kernel_mvt(double x1[120], double x2[120], double y_1[120], double y_2[120], double A[120][120]) {
 int i, j;

#pragma ACCEL PIPELINE flatten
#pragma ACCEL TILE FACTOR=1
#pragma ACCEL PARALLEL FACTOR=1
 for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x1 FACTOR=1
   for (j = 0; j < 120; j++) {
     x1[i] += A[i][j] * y_1[j];
   }
 }

#pragma ACCEL PIPELINE
#pragma ACCEL TILE FACTOR=1
#pragma ACCEL PARALLEL FACTOR=2
 for (i = 0; i < 120; i++) {
#pragma ACCEL PARALLEL reduction = x2 FACTOR=2
   for (j = 0; j < 120; j++) {
     x2[i] += A[j][i] * y_2[j];
   }
 }
}
```

Code 3: Code snippet of the `mvt` kernel with PIPELINE and PARALLEL pragmas as input to the Merlin Compiler.

```
// skipping the definition of helper modules for brevity
void mars_kernel_0_2_node_0_stage_0(int i,int exec,double _x2_buf_rdc_0[2][2]){
#pragma HLS INLINE OFF
```

```
  if (exec == 1) {
    merlinL6: for (int i_sub_0 = 0; i_sub_0 <= 1; ++i_sub_0) {
#pragma HLS unroll
      merlinL5: for (int j_sub_0 = 0; j_sub_0 <= 1; ++j_sub_0) {
#pragma HLS unroll
         _x2_buf_rdc_0[i_sub_0][j_sub_0] = ((double )0);
} } } }

void mars_kernel_0_2_node_1_stage_1(int i,int exec,double A_buf[120][120],double
      _x2_buf_rdc_0[2][2],double y_2_buf[60][2]) {
#pragma HLS INLINE OFF
  if (exec == 1) {
    merlinL9: for (int j = 0; j < 60; j++) {
#pragma HLS pipeline
      merlinL8: for (int i_sub = 0; i_sub < 2; ++i_sub) {
#pragma HLS unroll
        merlinL7: for (int j_sub = 0; j_sub < 2; ++j_sub) {
#pragma HLS unroll
          _x2_buf_rdc_0[i_sub][j_sub] += A_buf[((long )j) * 2L + ((long )j_sub)][((long )i) * 2L + ((long
                )i_sub)] * y_2_buf[j][j_sub];
} } } } }

void mars_kernel_0_2_node_2_stage_2(int i,int exec,double _x2_buf_rdc_0[2][2],double x2_buf[60][2]){
#pragma HLS INLINE OFF
 if (exec == 1) {
   merlinL11: for (int i_sub_1 = 0; i_sub_1 <= 1; ++i_sub_1) {
#pragma HLS unroll
     merlinL10: for (int j_sub_1 = 0; j_sub_1 <= 1; ++j_sub_1) {
#pragma HLS unroll
       x2_buf[i][i_sub_1] += _x2_buf_rdc_0[i_sub_1][j_sub_1];
} } } }

void mars_kernel_0_2(int mars_i,int mars_init,int mars_cond,double mars_A_buf_1[120][120],double
      mars__x2_buf_rdc_0_0[2][2],double mars__x2_buf_rdc_0_1[2][2],double
      mars__x2_buf_rdc_0_2[2][2],double mars_x2_buf_2[60][2],double mars_y_2_buf_1[60][2]) {
#pragma HLS INLINE OFF
  mars_kernel_0_2_node_0_stage_0(mars_i - 0,(int )(mars_i >= mars_init + 0 & mars_i <= mars_cond +
        0),mars__x2_buf_rdc_0_0);
  mars_kernel_0_2_node_1_stage_1(mars_i - 1,(int )(mars_i >= mars_init + 1 & mars_i <= mars_cond +
        1),mars_A_buf_1,mars__x2_buf_rdc_0_1,mars_y_2_buf_1);
  mars_kernel_0_2_node_2_stage_2(mars_i - 2,(int )(mars_i >= mars_init + 2 & mars_i <= mars_cond +
        2),mars__x2_buf_rdc_0_2,mars_x2_buf_2);
}

__kernel void kernel_mvt(double x1[120],double x2[120],double y_1[120],double y_2[120],merlin_uint_512
      A[1800]) {
#pragma HLS INTERFACE m_axi port=A offset=slave depth=1800 bundle=merlin_gmem_kernel_mvt_512_0
#pragma HLS INTERFACE m_axi port=x1 offset=slave depth=120 bundle=merlin_gmem_kernel_mvt_64_x1
#pragma HLS INTERFACE m_axi port=x2 offset=slave depth=120 bundle=merlin_gmem_kernel_mvt_64_x2
#pragma HLS INTERFACE m_axi port=y_1 offset=slave depth=120 bundle=merlin_gmem_kernel_mvt_64_0
#pragma HLS INTERFACE m_axi port=y_2 offset=slave depth=120 bundle=merlin_gmem_kernel_mvt_64_1

#pragma HLS INTERFACE s_axilite port=A bundle=control
#pragma HLS INTERFACE s_axilite port=x1 bundle=control
#pragma HLS INTERFACE s_axilite port=x2 bundle=control
#pragma HLS INTERFACE s_axilite port=y_1 bundle=control
#pragma HLS INTERFACE s_axilite port=y_2 bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

  double y_2_buf[60][2];
#pragma HLS array_partition variable=y_2_buf complete dim=2
  double x2_buf[60][2];
  double y_1_buf[120];
#pragma HLS array_partition variable=y_1_buf complete dim=1
  double x1_buf[120];
  double A_buf[120][120];
#pragma HLS array_partition variable=A_buf complete dim=2

  memcpy_wide_bus_read_double_2d_120_512(A_buf,0,0,(merlin_uint_512 *)A,(0 * 8),sizeof(double ) *
        ((unsigned long )14400L),14400L);
  merlin_memcpy_0(x1_buf,0,x1,0,sizeof(double ) * ((unsigned long )120),960UL);
  merlin_memcpy_1(y_1_buf,0,y_1,0,sizeof(double ) * ((unsigned long )120),960UL);

  merlinL15: for (int i = 0; i < 120; i++) {
#pragma HLS dependence variable=x1_buf array inter false
#pragma HLS pipeline
    merlinL14: for (int j = 0; j < 120; j++) {
#pragma HLS unroll
      x1_buf[i] += A_buf[i][j] * y_1_buf[j];
    }
  }

  merlin_memcpy_2(x1,0,x1_buf,0,sizeof(double ) * ((unsigned long )120),960UL);
  merlin_memcpy_3(x2_buf,0,0,x2,0,sizeof(double ) * ((unsigned long )120),960UL);
  merlin_memcpy_4(y_2_buf,0,0,y_2,0,sizeof(double ) * ((unsigned long )120),960UL);

  double _x2_buf_rdc_0[2][2];
  int mars_count_0_2 = 0;
  double mars_kernel_0_2__x2_buf_rdc_0_0[2][2];
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_0 complete dim=2
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_0 complete dim=1
  double mars_kernel_0_2__x2_buf_rdc_0_1[2][2];
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_1 complete dim=2
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_1 complete dim=1
  double mars_kernel_0_2__x2_buf_rdc_0_2[2][2];
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_2 complete dim=2
#pragma HLS array_partition variable=mars_kernel_0_2__x2_buf_rdc_0_2 complete dim=1

  merlinL13: for (int i = 0; i < 60 + 2; i++) {
  if (mars_count_0_2 == 0)
    mars_kernel_0_2(i,0,59,A_buf,mars_kernel_0_2__x2_buf_rdc_0_0,mars_kernel_0_2__x2_buf_rdc_0_1,
          mars_kernel_0_2__x2_buf_rdc_0_2,x2_buf,y_2_buf);
  else if (mars_count_0_2 == 1)
    mars_kernel_0_2(i,0,59,A_buf,mars_kernel_0_2__x2_buf_rdc_0_2,mars_kernel_0_2__x2_buf_rdc_0_0,
          mars_kernel_0_2__x2_buf_rdc_0_1,x2_buf,y_2_buf);
  else
    mars_kernel_0_2(i,0,59,A_buf,mars_kernel_0_2__x2_buf_rdc_0_1,mars_kernel_0_2__x2_buf_rdc_0_2,
          mars_kernel_0_2__x2_buf_rdc_0_0,x2_buf,y_2_buf);
  mars_count_0_2++;
  if (mars_count_0_2 == 3)
    mars_count_0_2 = 0;
  }
  merlin_memcpy_5(x2,0,x2_buf,0,0,sizeof(double ) * ((unsigned long )120),960UL);
}
```

Code 4: AMD/Xilinx Vitis HLS equivalent of Code 3 automatically generated by the Merlin Compiler.

Code 4 demonstrates the generated HLS code. The first nested loop is translated to loops merlinL15 and merlinL14 of the top-level function *kernel_mvt*. Because of the flatten pipelining, merlinL15 is pipelined while merlinL14 is completely unrolled. To satisfy this unrolling, the y_1_buf and A_buf buffers are partitioned in lines 232 and 235, respectively. It is worth noting that the required data are loaded before these loops and the resulting buffer, x1_buf is stored in off-chip memory after the computation is done.

The second nested loop is translated into the merlinL13 loop. The PARALLEL pragma in Line 156 of Code 3 produces a sub-loop with a trip-count of 60 (+2 because of pipelining) for the merlinL13 loop. The PIPELINE pragma in Line 154 divides the computation into three units, referred to as *mars_kernel_0_2_node_X_stage_X*, thus establishing coarse-grained pipelining in this context. Note that the inner-most loops of these modules have trip-counts of 2 because of the PARALLEL factor of 2 in lines 156 and 158.

References

[1] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. Source-to-source optimization for hls. pages 137–163, 2016.

[2] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for accelerator design and customized architectures. In IISWC, 2014.

[3] Tomofumi Yuki and Louis-Noël Pouchet. Polybench/c.