Tensor Optimization for High-Level Synthesis Design Flows

Marco Siracusa, and Fabrizio Ferrandi, *Member, IEEE,*

***Abstract*—Improving data locality of tensor data structures is a crucial optimization for maximizing the performance of Machine Learning and intensive Linear Algebra applications. While CPUs and GPUs improve data locality by means of automated caching mechanisms, FPGAs let the developer specify data structure allocation. Although this feature enables a high degree of customizability, the increasing complexity and memory footprint of modern applications prevent considering any manual approach to find an optimal allocation. For this reason, we propose a compiler optimization to automatically improve the tensor allocation of high-level software descriptions. The optimization is controlled by a flexible cost model that can be tuned by means of simple yet expressive callback functions. In this way, the user can tailor the optimization strategy with respect to the optimization goal. We tested our methodology integrating our optimization in the Bambu open-source HLS framework. In this setting, we achieved a 14% speedup on the digit recognition version proposed by the Rosetta benchmark. Moreover, we tested our optimization on the CHStone benchmark suite, achieving an average of 6% speedup. Finally, we applied our methodology on two industrial examples from the aerospace domain obtaining a 15% speedup. As a final step, we tested the versatility of our methodology inserting our optimization in the Clang software optimization flow achieving a 12% speedup on the Rosetta benchmark when running on CPU.**

I. Introduction

THE fast-paced evolution of machine learning techniques keeps posing ambitious performance requirements for hardware vendors. However, recent technical challenges in the semiconductor process required computer architects to investigate alternative solutions for a high-performance delivery. While Google proposed Tensor Processing Units (TPUs) [1], more flexible and cost-competitive approaches have been investigated. In this direction, Field-Programmable Gate Arrays (FPGAs) appear to be a promising solution [2][3]. In fact, despite FPGAs offer performances comparable to Application-Specific Integrated Circuits (ASICs), they require way reduced production costs and time to market.

FPGAs are integrated circuits providing programmable logic that can be configured to execute a certain function. The way FPGA devices can be configured substantially changed over time and nowadays a variety of choices is proposed. Despite in the past this choice had to be a tradeoff between performance and programmability, with the High-Level Synthesis (HLS) process [4] evolving over the years, nowadays FPGAs can be easily programmed through high-level languages with a negligible performance loss. High-Level Synthesis is a set of techniques used for producing a Register-Transfer Level (RTL) description starting from a high-level behavioral description. The obtained RTL description is then used for producing the bitstream responsible for programming the FPGA device. In general, during the HLS process, local arrays are allocated in Block RAM (BRAM) memories placed all around the programmable logic. Since tensors essentially are n-dimensional arrays, the same consideration applies. In principle, BRAM memories guarantee an access time in the order of clock cycles and an aggregate capacity ranging from kilobytes to megabytes according to the considered device. Despite this solution seems to provide ideal performance, there are two main issues to be considered. Firstly, each BRAM bank has a limited amount of ports to be shared among all of the data structures stored within. In case parallel accesses are required, other allocation strategies such as data partitioning or data reshaping should be investigated. Secondly, the too dense logic generated from large and complex designs might excessively stress the on-chip memory subsystem and compromise the overall performance. In fact, processing elements who need to access such memories are wired to the bank where the data to access is placed. In the case of large designs, long routings required to reach memory banks may congest the communication infrastructure causing severe performance degradations. Analogously to CPU and GPU architectures employing memory hierarchies [5] to improve data locality [6][7], these limitations could be overcome allocating critical data structures figuratively nearer the computational area. In this direction, the most promising solution consists of allocating highly-accessed tensors in the FPGA registers usually devoted to store scalar variables. In the HLS process, this optimization can be performed in the front-end layer with a compiler transformation disaggregating tensors in their scalar elements. Despite with a way different goal, the transformation enforcing data structure disaggregation has already been formulated by the software-compiler community and is known in the literature as Scalar Replacement Of Aggregates (SROA).

Since HLS tools [8] are usually built on top of software compilation frameworks such as GCC [9] and LLVM [10], several IR-level transformations can be inherited from the underlying compilation suite. However, since mainly CPU-specific, these optimizations may produce suboptimal results when applied to the HLS process [11]. Considering the SROA transformation, the CPU version is particularly focused on disaggregating structs types so that scalar-only optimizations could work on aggregate subelements too [12]. Firstly, this implies that CPU SROA has no real interest in aggressive interprocedural applicability where advanced analysis and trans-

This article was presented in the International Conference on Hardware/-Software Codesign and System Synthesis 2020 and appears as part of the ESWEEK-TCAD special issue.

M. Siracusa and F. Ferrandi are with the Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Milan, Italy.
E-mail: marco.siracusa@mail.polimi.it, fabrizio.ferrandi@polimi.it

formations are required. Secondly, the provided cost model would likely be quite far from HLS interests. In principle, while register-allocation reduces BRAM usage for general performance improvement, there are certain circumstances where this optimization should be controlled. In fact, FPGA registers are, in a sense, limited resources whose usage should be somehow regulated. Furthermore, considering that registers are non-addressable memory, accessing a register-allocated array with a non-constant index requires additional logic and latency to multiplex the data access. Aware of this issue, some commercial HLS tools allow to enforce register allocation of a selected array by means of HLS directives placed in the source code. Although this methodology enables a great deal of control, it does not scale with nowadays application's complexity where the user should explore too large optimization spaces to find an optimal allocation configuration. Moreover, this manual approach is not applicable when synthesizing Domain-Specific Languages providing no optimization expressiveness.

For this reason, we propose an open-source [1] HLS optimization for automatically optimizing tensor locality of Machine Learning and intensive Linear Algebra applications. This implementation particularly values effectiveness and portability. Effectiveness is managed proposing the optimization as a set of analyses and transformations to be strategically placed all over the optimization chain. In this way, we do not need to condense the full application complexity in a single bulky optimization rather allowing a modular and powerful solution actively interacting with other LLVM-builtin optimizations. Portability, instead, is guaranteed by a flexible cost model that can be customized by the user via simple callback functions. In practice, the optimization comes as a set of specific LLVM optimization and transformations mainly subdivided in preprocessing optimizations, function versioning, SROA extension, and code cleanup. Although possibly distributed in the optimization chain of standard compiler recipes (such as the *-O2* optimization level), all the transformations are working together since lead by the same cost model.

Despite we individuate machine learning as the main applicative domain, this methodology is generalized to any intensive linear algebra application. After being integrated into the Bambu [13] HLS framework, the proposed methodology has firstly been validated on one of the Rosetta benchmark [14] where we achieved a 14% speedup on the digit recognition test case. Moreover, we tested our optimization on a complex and HLS-standard scenario such as CHStone test suite [15] reaching a 6% average speedup. Then, we synthesized several industrial Aerospace applications obtaining a 15% improvement on these real case examples. Lastly, as a portability showcase, we customized the cost model to target CPU-based architectures and we illustrate the optimization of a sample application.

In definitive, we propose the following contributions:

1) we investigated the limitations of some CPU compiler optimizations when applied to HLS design flows;
2) we address those limitations proposing advanced analysis and transformations improving tensor allocation and accessing;
3) we integrated the analysis with callback functions to customize the cost model and tightly control the optimization goal;
4) we propose a second cost model for CPU, demonstrating the flexibility of our approach.

In the remainder of this document, in Section II we discuss related works and state-of-the-art methodologies in the HLS context. Then, in Section III we provide an LLVM overview to better discuss our methodology in Section IV. Finally, in Section V we validate our work on several benchmarks and industrial applications before summarizing and delineating future steps in Section VI.

II. Related Work

In the HLS process, tensors can be forced to registers allocation by means of the Scalar Replacement of Aggregates (SROA) IR-level compiler transformation. Despite this transformation is already implemented in the GCC [12] and LLVM [16] compiler suite, as shown by the results collected by Huang et al. [11], the current implementations do not provide any relevant performance impact in the HLS synthesis. More generally, the authors investigate the performance impact of several LLVM IR-level compiler optimizations when integrated into the front-end layer of an open-source HLS tool. As already discussed for software compilation [17][18][19], the collected results demonstrate how both IR-level optimizations' selection and ordering heavily impact the quality of the generated hardware description. Thus, despite a selected set of optimizations produces performance improvements, an aggressive CPU-specific standard optimization level such as -O3 degrades the overall HLS performance. In fact, since the set of optimizations offered by compilation suites such as GCC or LLVM mainly target CPU architectures, the used cost model may produce suboptimal results on other target architectures. In principle, CPU cost models trade optimization effectiveness for containing compilation time. Besides this limits its applicability in contexts like High-Level Synthesis where compilation time is not an issue [20], there may be cases where the transformation is anyways too aggressive and produce code patterns not recognized by the HLS phase, eventually leading to performance degradation. For this reason, Prabhakar et al. [21] propose a similar study yet considering a restricted subset of HLS-oriented optimizations. This work shows that HLS-specific optimizations provide substantial performance benefits even if requiring custom implementation and maintenance.

State-of-the-art HLS tools also implement HLS-specific optimizations that can be enforced with source-level HLS directives. In particular, data allocation can be managed with directives for *partitioning* and *reshaping* data structures and distributing array allocations over separate BRAM banks. In this way, the developer can increase memory access concurrency to support higher degrees of thread-level parallelism [22][23][24]. However, because of its nature, BRAM partitioning requires greater memory usage consequently introducing

[1] An LLVM implementation of the proposed optimization is publicly available at the following link https://github.com/ferrandi/PandA-bambu

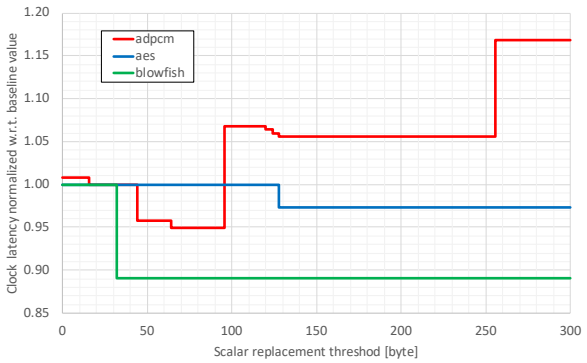

Fig. 1. Latency analysis of three Benchmarks from the CHStone test suite while varying disaggregation threshold of a commercial HLS tool.

higher design complexity. Since this aspect generally increases single-thread latency and timing, in case this loss is not amortized by the latency reduction brought by the parallel computation, the final design will suffer performance degradations. For this reason, in order to avoid BRAM allocation of critical data structures, HLS tools allow to completely partition arrays into single elements to be mapped in FPGA registers. In this way, the user can specifically improve the data locality of highly-accessed arrays for relevant performance improvement. However, since register allocation may come with substantial area and even latency cost, finding the optimal configuration requires further investigations.

Considering for instance the *adpcm* benchmark of the CHStone suite [15], since allocating a total of 21 arrays, assessing all the combinatorial effects requires to consider two million solutions in a full search approach. Although expert users are able to prune the optimization space, a manual search is still not feasible. For this reason, commercial HLS tools propose a threshold-based approach where all the arrays with size below a specific value are automatically lowered. For the sake of discussion, in Figure 1 we report the results we collected with a commercial HLS tool while varying the disaggregation threshold. The figure visualizes a clock latency analysis of three CHStone benchmarks obtained probing the whole threshold domain from 0 to 300 bytes. In particular, the Y-axis reports the circuit latency normalized with respect to the latency obtained with the default threshold value. Thus, values lower than one imply that there is at least one threshold value providing higher performance than the default one. Considering the *aes* and *blowfish* cases, there are two threshold values providing better results than the default one. However, being these values different, it requires the user to fine-tune the threshold for each application. Furthermore, the results obtained from *adpcm* demonstrates that the latency is not generally decreasing with increased disaggregation and then, since there are different local minima, no pruning strategy can be applied. Although this threshold-based approach provides a simplified method with respect to HLS-directive insertion, it drastically reduces the explorable optimization space. In fact, optimal solutions usually come from an arbitrary combination of arrays' disaggregation not considered by a threshold-based approach. Considering the *adpcm* example, the optimization space of the threshold-based approach reduces to the eight items represented by the vertical traits in Figure 1, hardly guaranteeing optimality.

For these reasons, we propose a tensor optimization for automatically improving the data locality of critical data structures. In particular, we mainly focus on IR-level tensor disaggregation performed through Scalar Replacement Of Aggregates (SROA) transformation. In order to improve SROA applicability in the HLS process, we perform a more complex and modular transformation with respect to the LLVM and GCC implementations, also including a set of accessory analyses and transformations improving optimization results. Moreover, we integrate all the optimizations with a flexible and customizable cost model. In this way, we provide an automated optimization currently missing in modern commercial and academic HLS tools. Providing high customizability by means of simple callback functions, the cost model can be easily adapted to other architectures.

III. LLVM Background

The LLVM infrastructure is a collection of compiler technologies used for optimizing high-level code through a modular toolchain. At the higher level, this toolchain is composed of a frontend, an optimization layer, and a backend. The frontend, commonly addressed as the Clang compiler, constitutes a tool for translating a large choice of high-level languages into a target-independent representation formally known as LLVM Intermediate Representation (LLVM IR). This intermediate representation is used in the optimization layer to perform target-independent optimizations. The optimized code is then forwarded to the backend layer that, after performing target-specific optimizations, translates it to machine code.

The LLVM IR is a strongly typed RISC instruction set used by the optimization layer to perform target-independent optimizations. In practice, the LLVM IR resembles an assembly-like language in the Static-Single-Assignment form. However, this language offers some higher-level constructs used for abstracting strong target-dependent concepts. For example, the complex mechanism behind the calling convention is abstracted through a simple *llvm::CallInst* statement representing a function call. The backend would then take care of any lowering according to the considered target. An LLVM program is an *llvm::Module* containing *llvm::GlobalVariable*s and *llvm::Function* definitions or declarations. Each *llvm::Function* contains several *llvm::BasicBlock*s defining the control flow graph. Each *llvm::BasicBlock* contains an ordered list of *llvm::Instruction*s to be executed in sequence. Using specific analysis, higher-level constructs such as *llvm::Loop*s can be tracked.

The LLVM IR is a powerful representation where the optimization layer can operate through a chain of *compiler passes*. Each pass is entitled to perform a certain *analysis* or *transformation* either on a *llvm::Module*, *llvm::Function*, *llvm::Loop* or *llvm::BasicBlock*. In practice, a pass is a C++ class that reads or manipulates the LLVM IR through the LLVM Core API libraries. Each pass can declare the list of other passes that requires or invalidates. The *llvm::PassManager* entity is then responsible to coherently schedule all the passes.

In the end, a set of optimization ca be gathered in *recipes* defining the optimization steps needed to accomplish a certain level of optimization. Well known standard optimization levels include -O1, -O2, -O3, -Oz and so on. Therefore, the developer who wants to implement a new optimization recipe is only required to define the set of transformations to be applied and the order those should be carried out. It is just worth mentioning that other than in the optimization layer, optimizations can be implemented either in the frontend or backend stage despite offering infrastructures way less featured.

IV. Proposed approach

Our approach mainly aims at extending the Scalar Replacement of Aggregates (SROA) transformation to improve optimization applicability and flexibility. In this direction, we propose an interprocedural implementation composed of several analyses and transformation modules. These modules are coordinated by a flexible and customizable cost model guaranteeing optimality convergence. For this reason, we start discussing the main critical aspects of the SROA transformation (Section IV-A) and then we detail the proposed optimization recipe (Section IV-B) for improving tensor locality.

A. SROA criticalities

The SROA transformation disaggregates compound data structures in their fist-class-type elements. Although in software compilation this transformation is used for improving the applicability of further optimizations [12], in High-Level Synthesis it has a much more relevant impact. From the IR-level perspective of HLS tools, SROA still untangles alias-related issues and simplifies the context where the next optimizations would operate on. However, besides the data locality aspect widely discussed already, SROA also improves the generated hardware representation simplifying and reducing the generated logic and removing the indexing latency in critical paths. Nonetheless, to obtain the best benefits, certain potential applicability limitations should be addressed.

For the sake of clarity, we provide a toy example in Figure 2 demonstrating the effect of the transformation on a really simple scenario we are going to incrementally expand. In

Original Code

```
int foo() {
  int t[2][2] = {0, 1,
                 2, 3};
  return t[0][0] + t[0][1]
       + t[1][2] + t[1][3];
}
```

After SROA

```
int foo() {
  int t0 = 0, t1 = 1,
      t2 = 2, t3 = 3;
  return t0 + t1
         t2 + t3;
}
```

Fig. 2. A simple example of tensor disaggregation through SROA.

principle, we should associate, where possible, any memory access to a tuple composed by the base address and the set of indexes identifying the accessed subelements. Base address and indexes are then used to identify the element to be accessed once the disaggregation takes place. However, there are cases where no exact mapping between base address and indexes can be found and the transformation for that specific aggregate gets inhibited. Pointer casts, conditional pointers, and complex pointer arithmetic are examples of operations preventing disaggregation. To avoid the transformation gets limited by those patterns, we should perform an initial *code preprocessing* phase. This phase is composed of a set of transformations taking care of canonicalizing, transforming or simplifying a certain aspect potentially preventing the disaggregation phase. However, there are chronic cases that pose irremediable limits inhibiting the transformation. A pragmatic case is when, for example, the aggregate is passed to an extern function we cannot operate on. This characteristic requires to have a mechanism to perform a feasibility check before starting the transformation. Moreover, there are cases where the transformation is feasible yet not worthy, requiring an additional profitability check only allowing promising expansions. Although we have no hope to operate on extern functions, we can work on internal ones implementing a context-dependent interprocedural optimization.

Despite an interprocedural optimization is way more effective, it introduces a great deal of complexity. Due to the C/C++ array to pointer conversion (array decay), we lose track of the first dimension of a function argument, information required for the disaggregation phase. Since we should expand a certain aggregate argument in its subelements, we need to know the exact number of elements the compound type should be disaggregated into. Figure 3 provides an example of an interprocedural disaggregation to be handled with a callgraph analysis to retrieve the arguments' dimension. In case a single

Original Code

```
int sum2(int *t) {
  return t[0] + t[1];
}

int foo() {
  int t[4] = {0, 1,
              2, 3};
  return sum2(&t[0])
       + sum2(&t[2]);
}
```

After SROA

```
int sum2(int t0,
         int t1) {
  return t0 + t1;
}
int foo() {
  int t0 = 0, t1 = 1,
      t2 = 2, t3 = 3;
  return sum2(t0, t1)
       + sum2(t2, t3);
}
```

Fig. 3. An example of interprocedural SROA.

function is called with different argument dimensions, we should replicate the function as many times as the number of required versions to ensure a unique function signature per function call. This technique is known in the literature as *function versioning* and an example is provided in Figure 4. In particular, we see how function *sumN(...)* should be replicated to be used with different tensors in *foo()*.

Original Code

```
int sumN(int *t,
         int dim);



int foo() {
  int t[4] = {0, 1,
              2, 3};
  return sumN(t, 2)
       + sumN(t, 3);
}
```

After Versioning

```
int sum2(int t0,
         int t1);
int sum3(int t0,
         int t1,
         int t3);
int foo() {
  int t0 = 0, t1 = 1,
      t2 = 2, t3 = 3;
  return sum2(t0, t1)
   + sum3(t0, t1, t2);
}
```

Fig. 4. An example of the function versioning.

However, while performing the versioning, each function call's signature should be computed considering the expand-

ability constraints and expandability profitability of each argument in its function scope. For example, assuming we cannot expand a certain tensor within a function scope but the same tensor is passed to a callee function which allows its expansion in its body, we can still expand the tensor in the callee function and create an interface between the not expanded aggregate in the caller function and the expanded argument in the callee. Through this method, we can isolate the constraints on the expandability and profitability within the function scope. An example of this behavior is shown in Figure 5.

Original Code

```
int sumN(int *t,
         int dim);
int foo() {
  int t[4] = {0, 1,
              2, 3};
  printf("%p", t);
  return sumN(t, 2);
}
```

After Versioning

```
int sum2(int t0,
         int t1);
int foo() {
  int t[4] = {0, 1,
               2, 3};
  printf("%p", t);
  return sum2(t[0],t[1]);
}
```

Fig. 5. An example of callee only disaggregation.

One of the main limiting factors in terms of profitability consists of accessing arrays by means of non-constant indexes. These accesses are handled multiplexing the array elements according to the unknown index. We provide a simple high-level example of this behavior in Figure 6. However, despite

Original Code

```
int t[2] = {0, 1};
void foo(int val,
         int idx) {
  t[idx] = val;
}
```

After SROA

```
int t0 = 0, t1 = 1;
void foo(int val,
         int idx) {
  switch(idx) {
    case 0:
      t0 = val;
    break;
    case 1:
      t1 = val;
    break;
  }
}
```

Fig. 6. An example of a non-constant disaggregation through SROA.

producing a relatively short latency, introducing an n-way multiplexer might turn quite expensive in terms of the consumed area. For this reason, disaggregation of elements accessed by non-constant indexes should be handled with care.

B. Proposed optimization recipe

The proposed tensor optimization comes as a chain of LLVM built-in and custom implemented analysis and transformations all pictured in Figure 7. The rationale of having a multi-stage optimization resides in the possibility of distributing the complexity of the transformation over the LLVM recipe providing high flexibility and customizability. This approach substantially differs from the standard implementation of a certain LLVM optimization pass usually performed in a single step. However, we found that the advantages of subdividing the optimization into several cooperating phases could not easily be achieved in the traditional way. In particular, from a high-level perspective, our recipe can be firstly subdivided into preprocessing transformations, extended SROA, and cleanup phase. Although working individually, all of these phases are

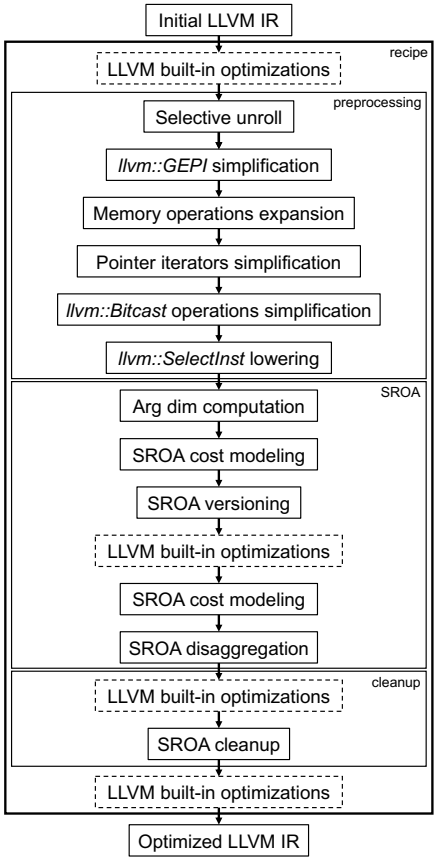

Fig. 7. LLVM recipe performing automatic tensor optimization.

coordinated by a single cost model defining the optimization goal. By means of simple user-defined callback functions, the optimization can be easily controlled and customized. Over this section, the technical aspects characterizing each phase are introduced along with a discussion about the cooperation and interoperation with those optimizations and some beneficial LLVM-builtin transformations we included in the recipe.

1) LLVM built-in optimizations: The LLVM infrastructure provides a large set of built-in analysis and transformations. Despite part of these optimizations are quite CPU-specific, a careful selection to be interleaved with our custom optimizations may substantially improve the quality of the generated code [11]. For this reason, our recipe considers different spots to include built-in optimizations. By means of a guided sensitivity analysis, we selected the most suitable optimizations to insert in each spot.

2) Selective unroll: Sine non-constantly indexed accesses may turn quite costly in terms of the area when performing tensor disaggregation, we try to reduce their overall amount with a controlled and selective loop unroll phase. In fact, since non-constant indexes mainly come from loops iterating over the tensor (as in Figure 8), an initial selective loop unroll phase can turn most of the indexes constant and allow propagation. However, it is important to notice that, if not controlled, loop unrolling could drastically change the resulting values of latency and area of the entire solution. Moreover, if we unroll a loop and the tensor accessed by the loop would not

be disaggregated, its elements residing in block memory could be the object of severe performance and area degradation. For this reason, through the usage of loop analysis [25], we make sure to only unroll loops providing benefits for the disaggregation phase and producing negligible movement in the Pareto solution. In detail, we control the unrolling through:

- the maximum number of iterations of the loop,
- the number of instructions contained in the loop,
- the estimated increase in resource usage of the loop,
- the arithmetical intensity of the loop.

In particular, the *arithmetical intensity* [6] of the loop is defined as the ratio between the *weighted instructions count* of the loop body over the number of bits accesses through aggregate data structures to be expanded in the disaggregation phase. In detail, we define the *weighted instructions count* of the loop body as the sum of the instructions appearing in the loop scope (excluding memory indexing) weighted by the number of bits they access and the area impact they have. However, for the sake of accuracy, we only count accesses through non-constant indexes and this index should be the induction variable of the loop. For example, with Figure 8 we provide a practical example on one loop performing an accumulation and another loop performing multiply-accumulate.

Loop Model Example

```
1  int sum_all(int t1[10], long t2[10]) {
2    int sum = 0;
3    loop1 : for(int i = 0; i < 10; ++i) {
4      sum += t1[i];
5    }
6    loop2 : for(int i = 0; i < 10; ++i) {
7      sum += t1[i] * (int)t2[i];
8    }
9  }
```

Fig. 8. An example of a loop the proposed cost model would unroll.

In particular, *loop1* would produce an arithmetical intensity of one integer sum over one integer access:

$$AI = \frac{32 * k_{sum\_weight}}{32 * k_{load\_weight}}$$

Conversely, *loop2* would produce an arithmetical intensity of one integer sum and one integer multiplication over one integer access and one long access resulting in an arithmetical intensity *AI* of:

$$AI = \frac{32 * k_{sum\_weight} + 32 * k_{mul\_weight}}{32 * k_{load\_weight} + 64 * k_{load\_weight}}$$

The operator weights, $k_{operator\_weight}$, have been defined by doing a regression on a large profiling dataset and they describe the different impact such operations have on the FPGA resource usage. For example, the k_{mul_weight} may have a value of 100 or more than the weight of k_{sum_weight} because using one DSPs over few DSPs available has a bigger impact than using 8 carry-chains to implement a sum. Measuring the arithmetical intensity of each loop ensures to unroll only memory-intensive loops more subject to benefits in the next disaggregation phase. The remaining non-constant indexed accesses untied by the loop unroll are expanded as an n-way multiplexer by the SROA optimization if meeting the profitability requirements.

3) llvm::GEPI simplification: In the LLVM IR, memory address computation is abstracted by means of the *llvm::GetElementPtrInst* (GEPI) [26]. This instruction computes the address of a subelement of an aggregate data structure starting from a base pointer and a set of indexes used to offset each aggregate dimension. In the backend of a software compiler, this abstraction is then lowered to a machine-level address computation considering the underlying memory layout and machine instruction set. For several reasons, certain address computations may be broken into several GEPIs generating potential replications. Although this redundancy has no much impact on software compilation, in the HLS process it may lead to instantiating useless operators and increasing associated area usage. For this reason, we implemented an optimization minimizing the overall number of GEPIs and simplifying the overall address computation.

4) Memory operations expansion: The LLVM language expresses specific memory operations such as *memcpy*s or *memset*s via intrinsic functions. However, instead of lowering these functions in the back-end HLS layer, we perform an early expansion so to facilitate our SROA transformation. For this reason, we replace the intrinsic function call with the actual logic implementing the called behavior, improving the next optimizations' applicability.

5) Pointer iterators simplification: In C/C++, array iteration can be performed either offsetting the base pointer with an index incremented at any iteration (i.e. `array[idx++]=k`) or directly incrementing the base pointer (i.e. `*array++=k`). Although the second form is slightly more efficient, it is less suitable for SROA and for the HLS process in general. In fact, without any explicit base address and dimension offsets, no SROA and other optimizations can be applied. For this reason, we transform, where possible, the second representation back to the first one. In practice, for each pointer iterator, we analyze its uses and look for a common base address and a set of indexes we can represent any access with.

6) Bitcast operations simplification: While CPUs and GPUs rely on NUMA memories with fixed bitwidth, FPGAs allow to configure their own on-chip memories. This implies that CPUs and GPUs, in order to maximize NUMA bandwidth, should fully exploit the memory bitwidth at each access. For this reason, CPU and GPU compiler implement local memory coalescing optimizations to maximize memory efficiency. For instance, if a computation fetches consecutive chars from memory, the compiler can coalesce the accesses and optimize memory transfer pulling eight chars at a time. These transformations can be easily individuated since using *llvm::Bitcast* operators. This operation is used in the LLVM intermediate representation to change the type of a memory pointer. In this way, in order to load a chunk of eight chars from memory, the compiler can bitcast the `*char` pointer to `*long` pointer and execute a single memory read. However, this optimization does not really comply FPGA memory management besides inhibiting SROA. For this reason, looking at the bitcasted type, we revert this transformation where possible.

7) llvm::SelectInst lowering: Analogously to pointer iterators masking index computation, *llvm::SelectInst* instructions may be an obstacle as well. This instruction is the LLVM

equivalent of the C/C++ ternary operator. Although quite useful, this operator may complicate the optimization context if the type of the selection is a pointer. Thus, we backtrack the selection path investigating whether we can express the given pointers in terms of a common base address and indexes.

8) Arg dim computation: In order to enable an interprocedural disaggregation, we should firstly individuate compound arguments' dimensions for further expansion. However, because of array decay occurring in C/C++, any argument can have different dimensions according to the function's call site and its context. For this reason, we perform a callgraph analysis recursively individuating the dimensions of each function call with respect to its calling context. The context is identified by the chain of function calls happening from the top function to the callsite currently analyzed.

9) SROA cost modeling: Our automated optimization strategy is based upon a custom-definable cost model. In practice, we estimate the profit of expanding a certain aggregate by means of a compiler analysis introduced in the recipe. In this way, each further transformation can query the analysis' results and take decisions accordingly. In principle, the expansion profit is computed summing up the effects on all the entities that would be affected by the aggregate expansion. The expansion effects on a given entity, instead, are returned from the associated callback functions modeling its behavior. For instance, considering the simple case in Figure 3, the expansion profit would mainly come from the indexing latency and area saved by the disaggregation. In the case of Figure 6, instead, the disaggregation requires to introduce a 2-way multiplexer. Although the introduced logic may result quite costly in terms of area, there could be cases where the overall latency reduction brought by the disaggregation makes the transformation still worthy. In order to formalize the cost model in terms of LLVM representation, we introduce the main LLVM entities [26] involved in the transformation:

- *llvm::AllocaInst*: the basic construct for allocating variables on the stack space;
- *llvm::GlobalVariable*: the basic construct for declaring a global variable;
- *llvm::Argument*: an argument of an *llvm::Function*;
- *llvm::GEPOperator*: the LLVM abstraction for performing indexing and field selection;
- *llvm::LoadInst*: the instruction used to load the content of a memory address in an SSA variable;
- *llvm::StoreInst*: the instruction used to store the content of an SSA variable in a memory address.

Starting from any *llvm::AllocaInst* and *llvm::GlobalVariable* having aggregate type, we recursively compute the disaggregation profitability as formalized in Table I. The function $profit(a)$ computes the profit of expanding an aggregate allocation (first formula) or declaration (second formula). The profit is computed summing up two components: the specific profit of the operator as given by the custom-definable callback function and the recursive profit computation of its *user*s. Concerning the first term, the callback functions are methods that the developer should overwrite to return a custom profit computation given a reference to the LLVM operation to be expanded and some context information regarding the operator. The developer can model the expansion profit using both the LLVM APIs to inspect the referenced operator and the additionally provided context information (such as the execution times of a given operator or the callgraph localization). For example, in the context of aggregate allocation or declaration, we implement a profit dependent on the aggregate size. In principle, our transformation migrates data structures allocation from block memories to registers, reducing memory congestion. For this reason, we decided that the transformation brings a linear revenue with respect to the allocated bits and a quadratic cost for inhibiting excessively large tensor's expansion. The exact values populating the callback models have been selected through a regression technique based on a large profiling dataset and sample applications. The second term of the allocation profit is given by the function $rec\_profit(a)$ recursively computing the profit of the allocation's users. According to the LLVM nomenclature, given an instruction or more generally a value, its *user*s are all the operations using that value as an operand. In particular, starting from the pointer given by an alloca instruction or global declaration, the $rec\_profit(a)$ function recurs over its users. In principle, the analysis recurs on the chained *getelementptr* instructions up to any memory operations (*load* or *store*) or callsite. In the case of memory operations, the analysis stops the recursion. In the specific case a callsite is encountered, the analysis proceeds in the function scope of the callee recursively computing the profit of expanding the associated argument. In case a different operator is encountered, a profit equal to $-\infty$ is returned, inhibiting any expansion for associated aggregate. For each iteration, we sum up the profit returned by the callback function associated with the involved operator. We implemented a $gepi\_callback$ returning a profit dependent on the nature of the GEPI indexes. Defining the profit as the revenue minus the cost, the constant indexed GEPIs provide a revenue proportional to the area saving associated with the expansion whereas the non-constant indexed GEPIs also provide a cost proportional to the logic required to multiplex the memory access. In this way, we inhibit non-constant indexed GEPIs' expansion which are however still possible if the global profit they bring motivates the disaggregation. Considering the example in Figure 3, we want to evaluate the profitability of expanding tensor `int t[4]`. Starting from its declaration in `foo`, the profitability is initialized to the value returned by the $alloca\_callback$. At the first step of the recursion, the analysis encounters a GEPI for computing the address of the operand `&t[0]` to be used in the callsite `sum2(&t[0])`. For this reason, the profit is updated querying the $gepi\_callback$. Since encountered a callsite, the analysis starts over computing the profitability cost of the argument associated with the callsite operand. Therefore, the analysis proceeds in the `sum2` function scope computing the expandability profit of the first argument. In this context, the profitability is updated considering the profitability associated with the two GEPIs for indexing the array access and the associated load instructions. Once returned from the function call, the second callsite (`sum2(&t[2])`) is analyzed in the very same way. Because of the reduced array dimension and the GEPIs having all constant indexes, the analysis returns positive profitability

$$\textbf{profit}(a\_alloc) = alloca\_callback(a\_alloc, context(a\_alloc)) + \sum_{u_{a\_alloc} \in users(a\_alloc)} rec\_profit(u_{a\_alloc}) \quad \text{if } a\_alloc \text{ is a alloca instruction}$$

$$\textbf{profit}(a\_glob) = global\_callback(a\_glob, context(a\_glob)) + \sum_{u_{a\_glob} \in users(a\_glob)} rec\_profit(u_{a\_glob}) \quad \text{if } a\_alloc \text{ is a global variable}$$

$$\textbf{rec\_profit}(a) = \begin{cases} gepi\_callback(a, context(a)) + \sum_{u_a \in users(a)} rec\_profit(u_a) & \text{if } a \text{ is a getelementptr instruction} \\ load\_callback(a, context(a)) & \text{if } a \text{ is a load instruction} \\ store\_callback(a, context(a)) & \text{if } a \text{ is a store instruction} \\ arg\_callback(a_i, context(a_i)) + \sum_{u_{a_i} \in users(a_i)} rec\_profit(u_{a_i}) & \text{if } a = op_i \text{ where } op_i \text{ is the operand } i \text{ of a callsite} \\ -\infty & \text{if } a \text{ is none of the listed operators} \end{cases}.$$

TABLE I
COST MODEL FORMALIZATION FOR COMPUTING TENSOR DISAGGREGATION PROFITABILITY

allowing the expansion. Conversely, in the case reported in Figure 6, the 2-way multiplexer introduced for allowing `int t[2]` disaggregation may inhibit its expansion if the function is not called enough times to motivate such area increase. A quite different scenario, instead, arises from Figure 5 where the `printf(...)` function turns the profitability of array `int t[4]` to $-\infty$. Although the base tensor is not expanded, the disaggregation of the array passed to function `sum2(...)` is still profitable, suggesting a callee only disaggregation.

10) SROA versioning: Versioning a function is generally quite costly in terms of area usage. For this reason, we should make sure to version a function only in case of need. Intuitively, a function should be versioned only if the profit of expanding its arguments is worthy with respect to the area increase. We compute the revenue of versioning a function summing up the expansion profit of its arguments previously computed by the cost model analysis. We compute the cost of versioning a function by modeling the area impact given by all of its versions. Computing the profit as revenue minus cost, we assess whether the given function should be versioned. In case the function is not versioned, the arrays passed as arguments would not be expanded in the SROA phase.

11) SROA disaggregation: This transformation performs the actual tensor expansion according to the results collected by the cost model analysis. In practice, the transformation recursively expands any *llvm::AllocaInst*, *llvm::GlobalVariable* and *llvm::Argument* having positive profitability. In doing so, we perform a preliminary analysis in order to individuate, for any memory access, the base address and the indexes used as dimensions' offset. This is performed starting from any load or store and recursively traverse the GEPI chain up to the base pointer (alloca instruction, global declaration, or function argument). The indexes of the traversed GEPIs are translated in aggregate dimensions offsets used for individuating the subelement to load or store. In case of variable dimension offsets given by non-constant indexes, we multiplex the access of that particular dimension by means of an n-way if-else-if construct whose condition is the variable index and the cases are the array elements. The preprocessing transformations take care of minimizing non-constant accesses and canonicalizing the GEPI chains to improve optimization applicability. In case, as in Figure 5, a callee-only disaggregation takes place, we set up the required interfaces.

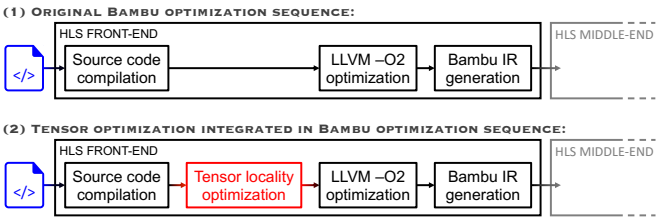

Fig. 9. Tensor locality optimization in the Bambu HLS framework.

12) SROA cleanup: We implemented a cleanup phase to perform SROA-specific simplifications. For instance, disaggregated function arguments are passed as pointers. However, if those pointers are read-only, we can pass them as references.

V. EXPERIMENTAL EVALUATION

In this section, we illustrate the impact of the proposed methodology integrating our optimization in Bambu [13], an open-source HLS tool. Bambu automatically generates hardware circuit implementations starting from C/C++ source code specifications. In particular, Bambu is composed of the front-end, middle-end and back-end layers. The frontend parses and optimizes the source code with LLVM or GCC and translates it to a different intermediate representation for the middle-end layer. Then, the middle-end layer performs hardware-specific optimizations before the actual HLS process is performed in the backend. As shown in Figure 9, we inserted our optimization recipe (listed in Figure 7) as a preprocessing phase of the LLVM optimization chain and we evaluated the performance of the original optimization chain (case 1 in Figure 9) versus the optimization chain integrating the tensor optimization (case 2 in Figure 9). It is essential to point out that the *-O2* optimization recipe originally used by Bambu already includes the SROA which, however, does not provide any relevant improvement. All of the experiments have been conducted on a Xilinx Virtex7 device (xc7vx690t-3ffg1930-VVD). Furthermore, in order to prove the flexibility of our approach, we modify the cost model of our optimization so as to be tested in a separate CPU compilation flow. The considered test cases are part of the Rosetta HLS benchmark suite [14], the CHStone benchmark suite [15] and several industrial applications in the aerospace domain. In particular,

we firstly consider the Rosetta benchmark for exemplifying how our optimization applies to fundamental machine learning applications. Then, through the CHStone test suite, we demonstrate how our optimization generalizes to other domains characterized by intensive linear algebra computation.

A. FPGA: The Rosetta benchmark suite

We selected a simple application from the Rosetta benchmark to be used as a walkthrough for the proposed approach. In this way, we aim at providing full details of the main concepts discussed in the previous sections. We selected the software version of the *digit recognition* algorithm for providing a detailed explanation of how our optimization works. The design performs hand-written digit classification through the K-Nearest-Neighbor (KNN) algorithm. We start considering that the main optimization opportunities arise from the `int dists[3]`, `int labels[3]` and `int votes[10]`. Despite `votes` only lives in the scope of function `knn_vote`, arrays `dists` and `labels` are propagated through function `update_knn` and `knn_vote`. This allows our optimization to independently expand those arrays in either the caller or callee, according to the configuration maximizing the profit.

The optimization begins with the *preprocessing phase* which transforms the code in a more HLS-oriented format favoring our next optimization phases. As discussed in Section IV, those optimizations are not meant to exploit processing element replication to improve parallelism yet are rather supposed to maximize the applicability of the proposed approach. In fact, unrolling loop `SET_KNN_SET` in `DigitRec_sw`, loop `FIND_MAX_DIST` in `update_knn` or the first loop in `knn_vote`, would avoid requiring a multiplexer for expanding the accessed tensors with no relevant increase in area usage.

The second phase of the proposed optimization considers the user-defined cost model to improve the tensor allocation for the underlying hardware. The goal of this phase is to lower tensors' degrading performance or area when considered as aggregate. In fact, registers are less constraining than block RAM and usually of large availability. In order to perform this phase correctly, the transformation needs to execute function versioning and disambiguate the function signatures compromised by the standard array to pointer conversion. In this simple example, all of the arguments have trackable dimension and no function should be disambiguated through replication. In case of need, the optimization first performs preliminary analysis on the tensor expansion profit and then assesses whether to duplicate the function according to its estimated cost. After function versioning, some optimizations such as constant propagation and dead code elimination are performed. At this point, the optimization of the tensor allocation takes place. This optimization uses the user-defined cost model to make considerations on the profitability of expanding, for example, the array `label` in the function `DigitRec_sw` it is declared or only in the scope of `update_knn` or `knn_vote` it is propagated. The profitability of expansion is derived from the cumulated profitability of the entities involved in the expansion. The profitability of a single entity is defined through user-defined callback functions called by the optimization during the reduction. The major contributions to profitability are given by the speedup and area impact a certain entity would provide. For example, `int votes[10]` in `knn_vote` looks a promising expansion but requires adding logic to the design for multiplexing the non-constant indexed accesses. For this reason, the code simplification phase is performed.

Once tensor lowering is performed, other specific transformations exploiting optimizations opportunities deriving from the previous phases are executed. For example, read-only pointer arguments are mutated to scalars or stored-once pointer arguments are moved as module return.

Through the described procedure, the proposed tensor optimization produces a 14% speedup without affecting resource usage. In particular, Table II compares the design produced by the original Bambu optimization sequence against the design produced by inserting our tensor optimization as a preliminary step as shown in Figure 9.

B. FPGA: The CHStone benchmark program suite

The CHStone benchmark suite offers a great variety of applications from several domains where we can show where and how our optimization better applies. Moreover, the proposed benchmarks offer a complex code structure for testing all of our optimizations and heuristics.

In order to assess the impact of our optimization, we firstly report in Table III the results in terms of performance and area obtained from the original optimization sequence of Bambu (case 1 in Figure 9). Then, we list in Table IV the results obtained after integrating our optimization in the original optimization chain (case 2 in Figure 9). As an added value, in Table V we report optimization statistics about the overall byte size of the involved tensors and the overall byte size of tensors our heuristic selected for optimization. A first comparison of the aggregated results shows that our optimization generally produces designs with reduced *Num Cycles* and higher *Frequencies*, leading to lower *Wall Clocks*. This behavior comes from the synergy of the different phases that the proposed transformation is composed of. In order to provide a better insight into the behavior of the proposed optimizations, we report in Figure 10 the Pareto walk of our transformation in optimizing three benchmarks of choice. The three selected benchmarks mean to be significant samples of various situations possibly arising during the overall tensor optimization. In particular, we compare how the different phases composing the proposed tensor optimization (pictured in orange) and the Bambu original optimization flow (pictured in blue) move in the performance-area space represented by the Pareto chart. The performance indicator we consider is the Wall Clock whereas the area consumption of interest is the number of slices used by the design. On one side, we represent the walk of the Bambu original optimization flow as a single step summarizing the whole optimizations performed. On the other side, we represent the walk of the proposed approach as a three-step path composed of the *code preprocessing*, *tensor disaggregation* and *-O2* optimization phase. The charts show how the optimization sequence we propose might take counterintuitive paths before converging to

Setting	Wall Clock [us]	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency [MHz]	Clock Slack
Bambu original opt seq	$5.558 \cdot 10^4$	9181934	927	340	545	1	2	165.21	3.95
Bambu with tensor opt	$4.804 \cdot 10^4$	6481814	923	334	500	1	2	134.92	2.59

TABLE II
PERFORMANCE OF DIGIT RECOGNITION WITHOUT AND WITH TENSOR OPTIMIZATION AS SHOWN IN CASE 1 AND 2 OF FIGURE 9 RESPECTIVELY.

Benchmark	Wall Clock [us]	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency [MHz]	Clock Slack	HLS Time [s]
adpcm	$2.248 \cdot 10^2$	15522	6240	2171	4752	77	14	69.05	0.52	55.18
aes	$3.393 \cdot 10^1$	2879	3526	1157	2164	0	8	84.86	3.22	25.99
blowfish	$1.070 \cdot 10^3$	92256	3282	1055	2184	0	14	86.21	3.40	13.19
dfadd	$2.652 \cdot 10^0$	210	1854	600	790	0	0	79.18	2.37	22.57
dfdiv	$2.315 \cdot 10^1$	1784	3055	978	1894	18	0	77.07	2.03	26.18
dfmul	$1.076 \cdot 10^0$	90	1115	381	635	10	0	83.61	3.04	18.98
dfsin	$6.660 \cdot 10^2$	45535	8495	2688	4157	31	0	68.37	0.37	59.44
gsm	$4.182 \cdot 10^1$	2885	3955	1232	2155	30	5	68.99	0.51	61.36
jpeg	$6.744 \cdot 10^3$	463839	13983	4721	8085	8	58	68.78	0.46	70.21
mips	$3.164 \cdot 10^1$	2496	946	296	411	3	4	78.88	2.32	9.90
mpeg2	$2.825 \cdot 10^1$	2201	6931	2586	5040	0	1	77.92	2.17	25.30
sha	$1.300 \cdot 10^3$	113318	1845	600	1369	0	12	87.15	3.53	7.37
Average								77.50	2.00	
Overall	$1.017 \cdot 10^4$	743015	55227	18465	33636	177	116			395.67

TABLE III
RESULTS OBTAINED FROM THE ORIGINAL OPTIMIZATION SEQUENCE OF BAMBU (CASE 1 IN FIGURE 9)

Benchmark	Wall Clock [us]	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency [MHz]	Clock Slack	HLS Time [s]
adpcm	$1.399 \cdot 10^2$	9583	10546	3507	8058	101	8	68.50	0.40	92.02
aes	$2.436 \cdot 10^1$	2116	3848	1397	2259	0	14	86.87	3.49	40.49
blowfish	$9.996 \cdot 10^2$	90435	3156	1045	2296	0	14	90.47	3.95	21.34
dfadd	$2.652 \cdot 10^0$	210	1854	600	790	0	0	79.18	2.37	33.51
dfdiv	$2.281 \cdot 10^1$	1784	3056	959	1894	18	0	78.20	2.21	39.65
dfmul	$1.076 \cdot 10^0$	90	1115	381	635	10	0	83.61	3.04	27.52
dfsin	$6.341 \cdot 10^2$	45535	8419	2631	4146	31	0	71.81	1.07	97.09
gsm	$3.557 \cdot 10^1$	2399	5241	1658	2642	35	1	67.45	0.18	135.92
jpeg	$6.808 \cdot 10^3$	474342	19805	6432	9791	9	58	69.68	0.65	129.62
mips	$2.061 \cdot 10^1$	2479	866	258	306	0	4	120.29	6.69	13.04
mpeg2	$2.487 \cdot 10^1$	2134	6418	2462	4644	0	1	85.79	3.34	18.72
sha	$8.505 \cdot 10^2$	106113	1711	533	1173	0	12	124.77	6.98	9.94
Average								85.56	2.87	
Overall	$9.564 \cdot 10^3$	737220	66035	21863	38634	204	112			658.86

TABLE IV
RESULTS OBTAINED INTEGRATING THE PROPOSED TENSOR OPTIMIZATION IN THE ORIGINAL OPTIMIZATION SEQUENCE OF BAMBU (CASE 2 IN FIGURE 9)

	adpcm	aes	blowfish	dfadd	dfdiv	dfmul	dfsin	gsm	jpeg	mips	mpeg2	sha
Total stack allocated tensors	0	304	112	32	32	32	32	426	4612	384	56	320
Total globally allocated tensors	3384	4542	18736	2158	1578	1530	1626	912	54006	240	6296	16496
Total function argument tensors	96	40	24	8	68	36	68	84	429	0	50	10
Optimized stack allocated tensors	0	16	16	0	0	0	0	54	0	0	8	0
Optimized globally allocated tensors	472	128	0	0	0	0	0	16	0	64	96	40
Optimized function argument tensors	96	0	8	0	0	0	0	76	0	0	16	0

TABLE V
SUMMARY OF TENSOR OPTIMIZATION STATISTICS

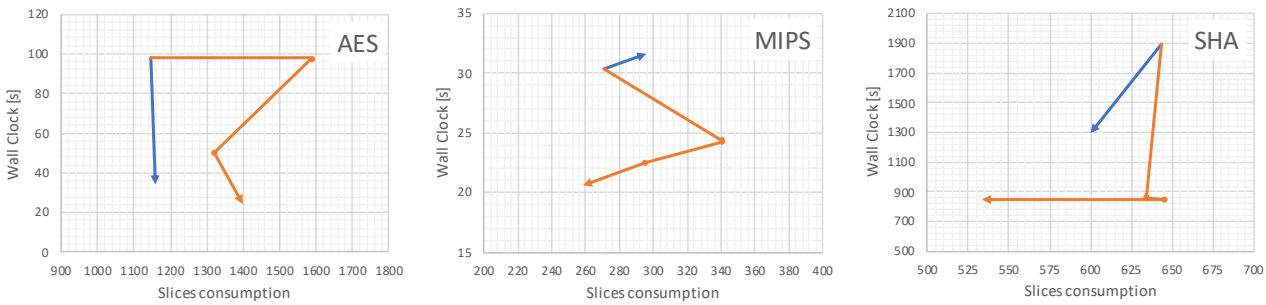

Fig. 10. Pareto walks of the proposed approach (orange line) and the Bambu original optimization sequence (blue line) on a CHStone benchmark selection.

performance improvement. In particular, the accuracy of the defined cost model determines the convergence to optimality. In fact, our optimization and the cost model it relies on do not aim at improving the performance in one step. Instead, as shown in Figure 10, they aim at transforming the code to be more suitable for the next phases and the HLS process. In particular, with a close look at Table IV we see that *jpeg* is the only benchmark whose performance are slightly worsened. However, *jpeg* presents characteristics quite difficult to be captured from a generic cost model, leading to slightly lower performance. In fact, *jpeg* requires extensive function versioning and expansion of memory accesses performed through non-constant indexes. Our cost model considers those transformations not profitable since requiring excessive area usage and therefore no main optimization is performed. The only consistent increase in area usage is only given by *adpcm* which approaches a 38% resource increase in order to allow a reduction of BRAM consumption of 43%. The other cases, instead, the cost model produces optimized designs with a difference in resource consumption ranging from a -15% (saving) up to a +26% with an overall average of +6%

C. FPGA: Industrial application in aerospace domain

Two different algorithms used by an aerospace company for Attitude and Orbit Control Systems have been investigated. Both the algorithms have been developed using a model-based design methodology exploiting Simulink and then Embedded coder, both from Mathworks, to translate the specification in C. The high-level synthesis of the C code was required to maintain the consistency between the C code running on the software side and the hardware implementation on the FPGA allowing to switch from software to hardware smoothly. The results obtained for the two algorithms are reported respectively in Tab. VI and Tab. VII where a comparison of the Bambu compilation flow with and without tensor optimizations is provided. Those examples are particularly relevant since our optimization produces improvements in any Pareto direction. In particular, in this set of two industrial benchmarks, the proposed tensor optimization simplifies the HLS process by performing SROA and function versioning of functions with a signature similar to the following one:

```
int32_T sMultiWordCmp(const uint32_T u1[],
        const uint32_T u2[], int32_T n);
```

This way to declare functions is usually a problem when commercial HLS synthesis tools are considered. The standard solution is to use a user estimated upper bound of the size of the arrays passed to the functions. Instead, the proposed approach discovers that in many places the number of elements of the passed arrays is limited and they can be effectively optimized by a combination of versioning and SROA steps.

D. CPU: software compilation flow integration example

We tested the versatility of our approach introducing our optimization in the CPU software compilation flow. In order to adopt our optimization to the CPU architecture, we removed strong-HLS-specific optimizations from the pass chain and reshaped the cost model according to the underlying hardware characteristics. In particular, we modified the callback functions defining the new cost model considering that:

1) non-constant indexed memory accesses should be avoided since frequent and atomic conditional branches generally degrade CPU performance
2) function versioning and unrolling should not be severely penalized since potentially leading to code simplification
3) tensor allocation should consider vectorization, calling conventions and memory-related issues as alignment

We tested the new setting on both the Rosetta and CHStone benchmark suites where we compiled the CPU code applying the tensor optimization before the *-O3* standard optimization sequence. The tighter constraints applied from the cost model reduced the optimization applicability to fewer cases of interest reported in Table VIII. In particular, we detail the results in terms of execution time on a 2.3 GHz Intel Core i5 architecture in two settings. In the first setting (baseline in Table VIII), we report the execution time before and after compiling the code with *-O3* (with disabled tensor optimization). The second setting (tensor optimization sequence in Table VIII), instead, performs tensor optimization just before the *-O3* recipe. For a better understanding, the partial results of *code preprocessing* and *tensor disaggregation* phases are reported.

VI. Conclusion

We presented a tensor optimization automatically reallocating the high-level data structures of a software description in order to better exploiting the underlying hardware characteristics. The modularity and customizability of this optimization allow a wide application on different domains and architectures. While we extensively discussed application in the HLS context, an extract of integration in a CPU context has been reported as proof of concept. As future work, we plan to extend the cost model computation with information forwarding from the frontend layer. In this way, advanced considerations can be done on the basis of features extracted from Domain-Specific Languages.

References

[1] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017.

[2] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *CoRR*, vol. abs/1901.00121, 2019.

[3] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA'17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 5–14.

[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for FPGAs: From prototyping to deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, 2011.

[5] C. Lameter, "NUMA (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, pp. 40–51, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2508834.2513149

[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

Setting	Wall Clock [us]	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency [MHz]	Clock Slack
Bambu original opt seq	45.394	1212	23138	7575	7394	111	16	26.7	2.55
Bambu with tensor opt	38.322	1127	23302	7325	7525	71	1	29.4	6.00

TABLE VI
PERFORMANCE OF AOCS1 WITH AND WITHOUT TENSOR OPTIMIZATION (25MHZ DESIGN CLOCK CONSTRAINT)

Setting	Wall Clock [us]	Num Cycles	LUTs	Slices	Registers	DSPs	BRAMs	Frequency [MHz]	Clock Slack
Bambu original opt seq	12.047	1244	23111	7615	12086	181	16	103.3	0.32
Bambu with tensor opt	11.069	1138	22637	7542	11693	172	1	102.8	0.28

TABLE VII
PERFORMANCE OF AOCS2 WITH AND WITHOUT TENSOR OPTIMIZATION (100MHZ DESIGN CLOCK CONSTRAINT)

	baseline [usec]		**tensor optimization sequence [usec]**		
	init	*O3*	*code preprocessing*	*tensor disaggregation*	*O3*
3Drender	11216	4665	9852	7516	3196
blowfish	274.5	134.8	270.3	260.2	123.7
mpeg2	11.93	0.190	7.339	7.151	0.146

TABLE VIII
TENSOR OPTIMIZATION ON SOFTWARE COMPILATION

[7] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat *et al.*, "Trends in data locality abstractions for HPC systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 3007–3020, 2017.

[8] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2016.

[9] R. M. Stallman and G. DeveloperCommunity, *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. Scotts Valley, CA: CreateSpace, 2009.

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," San Jose, CA, USA, Mar 2004.

[11] Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis for FPGAs," in *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, 2013, pp. 89–96.

[12] M. Jambor, "The new intraprocedural scalar replacement of aggregates," https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=get&target=jambor.pdf, 2010, accessed: Apr 8 2020.

[13] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4.

[14] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin *et al.*, "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software-Programmable FPGAs," *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, Feb 2018.

[15] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[16] "Scalar replacement of aggregates class reference," https://llvm.org/doxygen/classllvm_1_1SROA.html, accessed: Apr 8 2020.

[17] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO'03. USA: IEEE Computer Society, 2003, pp. 204–215.

[18] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves *et al.*, "Finding effective compilation sequences," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES'04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 231–239.

[19] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle, "Milepost GCC: Machine learning enabled self-tuning compiler," *International Journal of Parallel Programming*, vol. 39, pp. 296–327, 06 2011.

[20] M. Lattuada and F. Ferrandi, "A design flow engine for the support of customized dynamic high level synthesis flows," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 4, Oct. 2019.

[21] J. Cong, B. Liu, R. Prabhakar, and P. Zhang, "A study on the impact of compiler optimizations on high-level synthesis," in *Languages and Compilers for Parallel Computing*, H. Kasahara and K. Kimura, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 143–157.

[22] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications," pp. 430–437, Nov 2017.

[23] Y. Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

[24] M. Siracusa, M. Rabozzi *et al.*, "Automated design space exploration and roofline analysis for FPGA-based HLS applications," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 314–314.

[25] J. Liao, W.-F. Wong, and T. Mitra, "A model for hardware realization of kernel loops," in *Field Programmable Logic and Application*, P. Y. K. Cheung and G. A. Constantinides, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 334–344.

[26] "LLVM language reference manual," https://llvm.org/docs/LangRef.html, accessed: Apr 8 2020.

Marco Siracusa received his Bachelor's Degree in Computer, Electronic and Telecommunication Engineering in 2016 from the Università degli studi di Parma, Italy. He received a Master's degree in Computer Science and Engineering in 2020 from Politecnico di Milano, Italy. His research interests include Compiler Infrastructures, High-Level Synthesis, Computer Architectures and High-Performance Computing.

Fabrizio Ferrandi (M'95) received his Laurea (cum laude) in Electronic Engineering in 1992 and the Ph.D. degree in Information and Automation Engineering (Computer Engineering) from the Politecnico di Milano, Italy, in 1997. He has been an Assistant Professor at the Politecnico di Milano, until 2002. Currently, he is an Associate Professor at the Dipartimento di Elettronica, Informazione e Bioingegneria of the Politecnico di Milano. His research interests include synthesis, verification simulation and testing of digital circuits and systems. Fabrizio Ferrandi is a Member is a member of the IEEE Computer Society since 1995, the Test Technology Technical Committee, and the European Design and Automation Association.