Integrating Quick Resource Estimators in Hardware Construction Framework for Design Space Exploration

Bruno Ferres, Olivier Muller, Frédéric Rousseau

Integrating Quick Resource Estimators in Hardware Construction Framework for Design Space Exploration

Bruno Ferres, Olivier Muller and Frédéric Rousseau
Univ. Grenoble Alpes, CNRS
Grenoble INP†, TIMA,
F-38000 Grenoble, France
Email: {name.surname}@univ-grenoble-alpes.fr

***Abstract*—Hardware design processes often come with time-consuming iteration loops, as feedbacks generally result of long synthesis runs. It is even more true when multiple different implementations need to be compared to perform Design Space Exploration (DSE). In order to accelerate such flows and increase agility of developers — closing the gap with software development methodologies — we propose to use quick feedback generating transforms based on RTL circuit analysis for quicker convergence of exploration. We also introduce an Hardware Construction Language (HCL) based methodology to build explorable circuit generators, and demonstrate such usage over a General Matrix Multiply (GEMM) Chisel implementation. We demonstrates that using RTL estimation early in the exploration process results in $\times$7 less synthesis runs and $\times$4.1 faster convergence than an exhaustive synthesis process, and still achieves state of the art performances when targetting a Xilinx VC709 FPGA.**

Keywords: HCL - Chisel - FPGA - DSE - estimation - GEMM

I. Introduction

To build a digital design for a particular use case, hardware developers have multiple options for implementing functionalities, and it relies on their expertise to select and implement optimal choices in the process. To ease their work and reduce potential errors, Design Space Exploration (**DSE**) is commonly used to find a best fit among equivalent implementations, with respect to situation specific constraints and objectives. Automatic DSE tools often rely on design space partitioning between two sub spaces: potentially optimal solution — with respect to problem objectives — and sub optimal solutions. This first partition is composed of solutions which are estimated to be optimal, meaning that selecting another solution in their neighbourhood will either result in cost increase or performance decrease: this is known as the **Pareto curve** [1]. Such partitioning may not require to individually evaluate each solution, as exploration strategies can approximate this Pareto frontier without exhaustive space evaluation, resulting in faster completion of the exploration process. With respect to this statement, we highlight two parameters that can be leveraged for quicker exploration convergence: speed of the evaluation process, and space traversal strategy.

†Institute of Engineering Univ. Grenoble Alpes

However, many DSE tools rely on automatic inferences to generate different implementations, which can result in a lack of controllability over generated designs. To address this problem, one can use Hardware Construction Languages (**HCL**) — an emerging paradigm allowing users to control generated hardware by defining **hardware generators** instead of hardware circuits. Such feature eases code re-usability by providing higher level of programmability over circuits, while maintaining sufficient control over generated designs. As an implementation candidate, Chisel is an emerging HCL accepted by both industrial and academic world [2], and this work is based on it. For better apprehension of the hardware generator concept, a code snippet is introduced in Figure 1, where a **type parametric adder generator** is described, allowing instantiation of different adder modules. As we can see, HCLs enable to define generic data types — here we define a type `T` which inherits from `Data` and implements `Num[T]` (line 3), meaning it is an arithmetic type defining multiplications, additions, ... — and to instantiate it with different types — either 32 bit wide unsigned integers (line 14) or 8 bit wide fixed points type with 3 bits of precision (line 15), both types being defined in Chisel library (lines 0-1).

Nonetheless, DSE support for HCL initiatives are still to be proposed, and we present a Chisel based methodology as a *proof-of-concept*. We aim to speed-up exploration processes using two levers: we introduce quick resource usage estimators based on high level abstractions in Section III, and instrument a Chisel-based General Matrix Multiply (**GEMM**) module generator in Section IV, in order to compare two exploration strategies to find the "best" GEMM implementation for a given set of constraints. Section V then exposes experiments and results used to exhibit usability of such methodology, and Section VI presents conclusions and future works.

II. Related work

In order to perform efficient design space exploration, one needs to use **performant estimators**, *i.e.* estimators with an acceptable trade-off between speed and accuracy.

Methodologies to generate fast Register-Transfer Level (**RTL**) estimators for FPGA — such as resource usage, timing or even power consumption — have emerged this

```
0  import chisel3._
1  import chisel3.experimental.FixedPoint
2
3  class Adder[T <: Data with Num[T]](tpe: T)
4      extends Module {
5      val io = IO(new Bundle({
6          val op1 = Input(tpe)
7          val op2 = Input(tpe)
8          val res = Output(tpe)
9      })
10
11     io.res := io.op1 + io.op2
12 }
13
14 val uintAdder = new Adder(UInt(32.W))
15 val fpAdder = new Adder(FixedPoint(8.W, 3.BP))
```

Fig. 1: Type parametric adder generator in Chisel

past decades, using algorithmic descriptions as entry points, like Matlab [3][4][5], or C-based High-Level Synthesis (**HLS**) approaches. Such techniques recently grew in maturity, and need for performant estimators resulted in various techniques operating on Intermediate Representations (**IR**) — typically, Directed Dataflow Graph or other algorithmic IR — to quickly estimate both circuit performance and cost, allowing efficient exploration processes to be designed. Moreover, as no generic exploration strategy can be defined [1], different metrics need to be considered to perform circuit generation, varying from "classic" ones — *i.e.* area, latency, frequency, power consumption, ... — to more specific ones, such as quality-of-result [6], global throughput [7] or even multiplication mapping to Digital Signal Processors (**DSP**) for efficient primitive usage [8], resulting in sometimes use case specific estimation frameworks. In addition to that, resource usage does not grow linearly with circuit complexity, meaning that estimation methodologies may not scale, and some initiatives hence use statistical and/or machine learning approaches to build efficient estimators for HLS [9][10][11] or Domain-Specific Languages (**DSL**) [12] based explorations. HLS frameworks then uses high level transforms based on those quick feedbacks to perform efficient exploration [13], and try to approximate Pareto frontier to keep the number of estimated designs low [12][14]. However, all those approaches aim at reproducing synthesis tool behaviour while reducing flow duration, and are thus based on target specific characterization and optimization which may not be easily configurable by users — and entry point (HLS/DSL) may not fit every usage. HCL based initiatives are still to be proposed, even if some work already explored the opportunity they can bring for DSE — Schmidt et al. [15] proposed **JackHammer** to explore Secure Hashing Algorithm (SHA3) implementations, however they claim that the tool is too application-specific to be used as a generic solution, and it has not been made available yet.

On the other hand, initiatives for Hardware Description Language (**HDL**) based estimation have been explored to provide quick feedbacks for hardware developers. One of the main challenge when building such estimators is to cope with target specific optimizations in circuit generation, as different FPGA will not use the same resources for kernel implementation. To address this, two approaches can be considered: either allow user to define target specific optimization before estimation [16], or build estimators for a given board/family [17]. This last work aims to be integrated in any RTL based flow — as it is integrated in Xilinx PlanAhead tool — and relies on modelling synthesis steps instead of actually performing it. It identifies operation primitives and more complex macros using pattern recognition, performs target specific and non-specific optimization (*e.g.* operation merging or memory management) and then uses prior characterization results to generate accurate estimation.

As for the use case, General Matrix Multiply (**GEMM**) algorithm is a standard BLAS (*Basic Linear Algebra Subprograms*) routine, considered representative of algebra computations [18]. Implementation heavily relies on architecture choices — *e.g.* computation architecture, memory partitioning, communication protocol, ... — and can be used for more complex usages, such as Convolutional Neural Network (**CNN**) implementation[19][20][21] with heavy design space to explore [12][19].

In the context of our work, we aim at providing an HCL based design space exploration example, as an alternative to HLS based techniques — which result in less controllable hardware and widest design spaces. We chose to implement a generic estimation technique with principle similar to Schumacher et al. [17] and integrate it in standard Chisel flow as a *proof-of-concept*. We then used a Chisel based GEMM circuit generator to demonstrate our novel exploration methodology, based on previous work from Ferres et al. [22].

III. FIRRTL Based Resource Estimation

Chisel flow is based on a generic Intermediate Representation named **FIRRTL** (*Flexible Intermediate Representation for RTL*) [23][24], which is used to perform compilation optimization and verification — such as combinatorial loop checking, type inference, width reduction or dead code elimination — and Verilog generation. In order to integrate quick resource estimators in the flow, we thus chose to integrate in FIRRTL transform system — used for compilation passes — and define our estimators as simple transforms over the intermediate representation of the circuit.

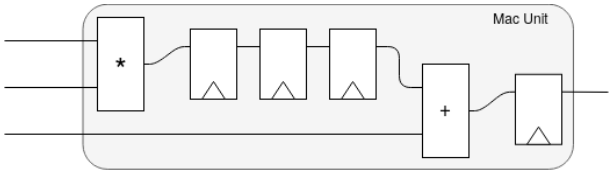
As resource estimation aims to approach vendors synthesis tools results, it is obviously target specific, and we will only consider target boards belonging to **Xilinx Virtex 7 family** in this work. However, estimation methodology is thought to be generic and one can easily build new estimators for a given target board.

A. Basic operator composition

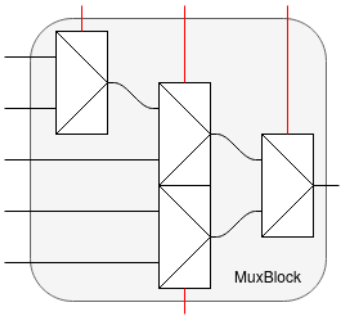
In order to accurately estimate the resource usage of a given design, our first approach was to individually estimate resource usage of each operator of the circuit, and add every individual estimation to obtain the global resource usage. Using FIRRTL hierarchy system, each submodule can be estimated only once, and global estimation is then performed using submodule instantiation amount.

As for the operator resource usage, we chose to only consider **eight operator kinds** as consuming resources: `ADD`,

MULT, BINOP, MUX, COMPARE, DSHIFT[1], REGISTER and MEMORY. Every other operator — *e.g. static shift, static padding, sub-word assignment* — are considered to consume zero resource as a first approximation, as they are basically only wires in data paths. For every operator consuming resources, we defined a simple Chisel Module, using input bitwidth as the only module parameter[2], and performed synthesis runs for every bitwidth in a given **bitwidth set**, and for a given **target board**. We then performed synthesis result analysis to extract resource usage for every implementation — considering 4 different resources: LUTs, Flip Flops, DSPs and BRAM — in order to build target specific libraries of characterized operators, then using linear interpolation to estimate resource usage for a given bitwidth. For REGISTER and MEMORY primitives, we do not need to perform preliminary syntheses, as we can count every FIRRTL register and memory instances instead. Of course, library building need to be done only once for a given target board — assuming we do not want to add an operator — and are stored in **JSON files**. They can then be used in the formerly described process to naively estimate circuit resource usage in a hierarchical manner, without performing synthesis over the whole circuit.



(a) Example of MAC unit found with $[\times\{Reg\}^{0..3}\{+\}^{0..1}\{Reg\}^{0..1}]$ pattern



(b) *n-to-1* multiplexers are estimated using $f(max(width_{input}), \#_{cond})$

Fig. 2: Macro block replacement

B. Macro block replacement

Even though this first approach is quite simple and straightforward, it cannot be applied directly for FPGA estimation.

As a matter of fact, FPGA inherent heterogeneous composition needs to be considered at synthesis time, when translating data paths to actual components: for example, adding a given signal with the result of a multiplication might be mapped to DSP instead of LUT for a simple addition, and such heterogeneity should also be considered when estimating resource usage, in order to approach a realistic estimation.

To do so, we provide a mechanism to describe and characterize macro blocks in way similar to the one described in Section III-A, along with a pattern recognition helper to find and replace given patterns in the FIRRTL representation. Figure 2 exhibits two different macro blocks that are being searched for when performing estimation: Multiply-and-Accumulate (**MAC**) units and multiplexers.

As stated earlier, MAC units should be considered as single blocks to approach synthesis results. Given Xilinx Virtex 7 specification, we know that DSPs can absorb up to 4 registers [25], and thus define replacement pattern of Figure 2a. This means that we will search through FIRRTL representation to find any data path where at least one multiplier is followed by 0 to 3 registers, 0 or 1 adder and 0 or 1 register, and replace it by a **Mac Unit macro block** parametrized by both input bitwidth and pattern parameters.

Another limitation is that FIRRTL representation only considers binary multiplexers — meaning that more complex multiplexers will be considered as a succession of *2-to-1* multiplexers, instead of *n-to-1*. As Xilinx Virtex 7 FPGA slices include *8-to-1* multiplexers — and allow more complex combination between slices — multiplexer resources can not be estimated accurately if *n-to-1* multiplexers are not considered in resource estimators, and we thus defined **MuxBlock macro block** (Fig. 2b).

It is important to remark that for multiplexers, we can not only consider macro block input bitwidth for parametrization, as the actual implementation also depends on the amount of conditions used for selection. We thus need to use multi-dimensional linear interpolation for estimation, using the 4 nearest synthesized mux blocks in the library to extrapolate resource usage of a given pattern.

Finally — and for the same reasons — memory primitives should be characterized using two factors: depth and width — as considering only total bit count in memory can not express the distributed aspect of embedded memories. We hence define a third macro block, **Memory macro block**, to address this.

Using basic operator characterization and macro block replacement, we hence build FIRRTL based resource estimators that are fully integrated in the Chisel standard flow.

IV. Exploring GEMM implementations

As a way to demonstrate the usage of such estimators for DSE processes, we chose to explore a General Matrix Multiply (**GEMM**) circuit generator already introduced in previous work [22]. This GEMM generator aims at **maximizing I/O throughput** by consuming input data only once, and we hence aim at exploring implementations on such criterion.

A. Defining design space

The first thing to do when building a DSE process is to define the explored design space.

In HLS — and some DSL [12] — processes, exploration spaces are defined using **knobs** [1], which can be seen as

[1]Dynamic Shift.

[2]In this work, we only considered unsigned integers, but characterization can be done on any Chisel arithmetic data type — even custom ones.

exploration parameters that will each define an exploration dimension. With this definition, the actual exploration space is built as the **Cartesian product** of the knobs value sets, and can result in very wide design spaces [12], highlighting the need for efficient exploration strategies.

In a similar way, we exposed **high level parameters** at top level in our Chisel `Module` to build our exploration space. As Chisel is based on scala, we used Scala's annotation system to embed parameters value sets directly in `Module` constructor, instead of using external configuration file to define exploration space. We defined 3 annotation kinds, which defines value sets for an **integer parameter p**:

- `@linear (x, y)` $\Leftrightarrow$ $p \in [\![x, y]\!]$
- `@enum (x0, ..., xn)` $\Leftrightarrow$ $p \in \{x_0, ..., x_n\} \wedge \forall i \in [\![0, n]\!], x_i \in \mathbb{N}$
- `@pow2 (x, y)` $\Leftrightarrow$ $p \in \mathbb{N} / p = 2^z \wedge z \in [\![x, y]\!]$

We also defined an `Explorable` trait based on scala reflective programming features, to build each different implementation in the Cartesian product of defined parameters by analysing `Module` constructor to retrieve exploration space.

```
class GemmRole(
    @pow2(5, 10) bandWidth: Int,
    @enum(32)    elemWidth: Int,
    @pow2(4, 10) dimension: Int
) extends Module with Explorable
```

Fig. 3: Exposing GEMM design space

For GEMM exploration, we defined design space (Fig. 3) using only 2 parameters — I/O bandwidth, and matrix dimension — while fixing element bitwidth to 32 bits, as it will not only impact architecture generation, but also algorithm quality of result. Varying I/O bandwidth over `@pow2 (5, 10)` — *i.e.* between $2^5 = 32$ and $2^{10} = 1024$ (6 values) — and dimension over `@pow2 (4, 10)` — *i.e.* between $2^4 = 16$ and $2^{10} = 1024$ (7 values) — we create a $6 \times 7 = 42$ **wide exploration space** to iterate over.

It is important to note that not every implementation in this design space is possible, since some combination of parameters may not be compatible for design generation — *e.g.* for Figure 3, one implementation among 42 is unfeasible: if *bandWidth* = 1024, *elemWidth* = 32 and *dimension* = 16, we got 32 elements sent each cycle ($\frac{1024}{32}$), but only 16 elements by line/column. As kernel design does not allow more than one line/column to be sent at once, this design will fail Chisel elaboration and will not be considered further in the flow.

B. Exploration strategy

After defining exploration space, one needs to define space traversal order, also known as **exploration strategy** — and it is even more important to define a clever strategy when the design space is too wide to be explored exhaustively, as it is the case in standard HLS flows.

First of all, we define the **pruning** of a design space as the exhaustive application of both an estimator and a pruning function over the given set of points — *i.e.* it consists in evaluating every point in the input design space, and accept or reject each of them, whether they fit or not the pruning function. Figure 4 shows an example of 2-Dimensional pruning, where the pruning function may be defined using the following equation, given imp_i an implementation in the input design space:

$$P(imp_i) = (Metric_A(imp_i) > 80) \vee (Metric_B(imp_i) > 50) \quad (1)$$

In this example, design space resulting from pruning will only include implementations for which $Metric_A < 80\ U_a$ **AND** $Metric_B < 50\ U_b$.

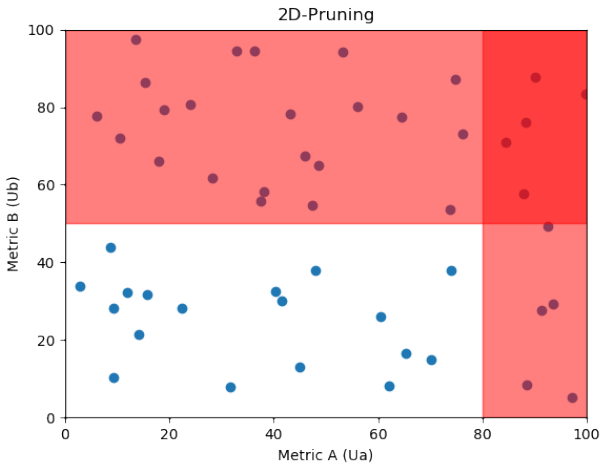


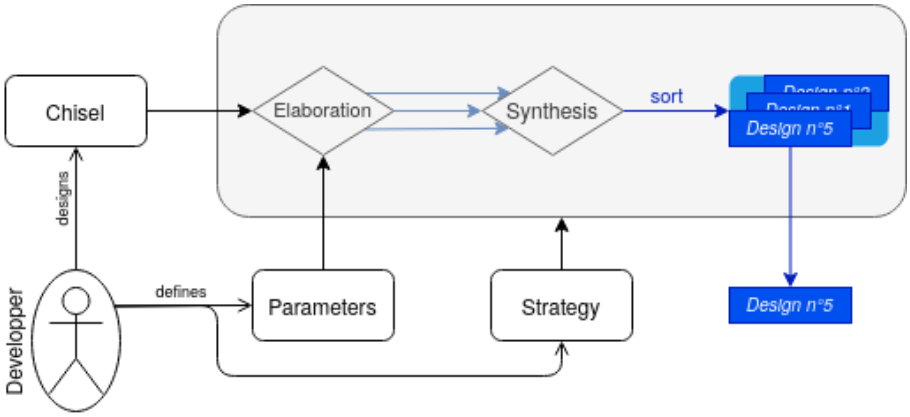
Fig. 4: Example of 2-Dimensional space pruning

We then propose to compare two *ad-hoc* strategies to demonstrate the usability of fast FIRRTL based resource estimation for faster DSE convergence:

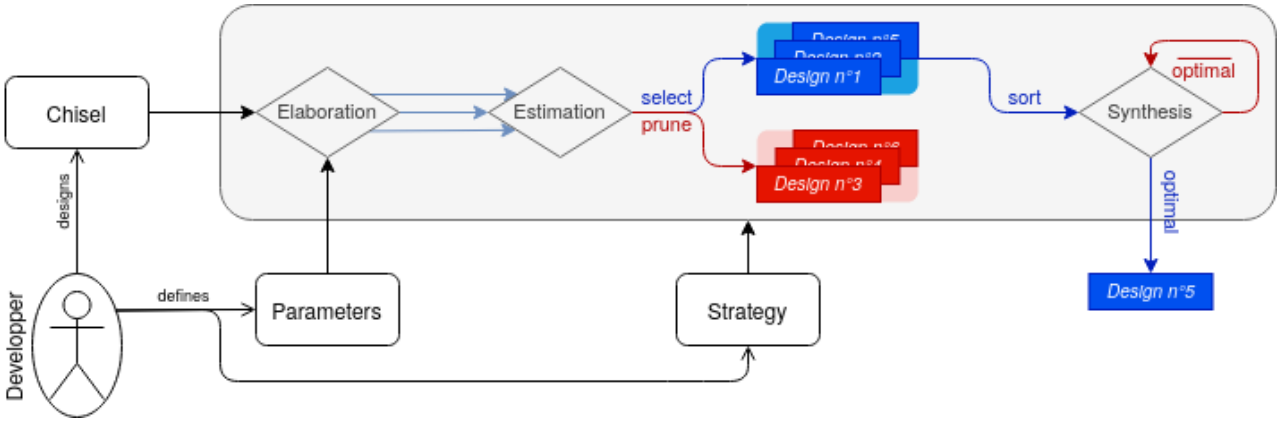
- exhaustive synthesis strategy, where every possible implementation is synthesized and compared (Fig. 5a)
- gradient based strategy, based on three consecutive steps (Fig. 5b):
  - estimate resource usage and **prune implementations** that consume **too many resources**[3]
  - sort design space to **select widest implementation** that still fits the target board, after this first pruning
  - use this candidate as the starting point of a **gradient descent algorithm** based on synthesis runs, aiming to find local optimum in an already pruned space

Both strategies are manually defined using **scala** to iterate and find best fits over every possible implementations, using defined estimators for fast resource estimation, and synthesis result parsing for more accurate ones. Users (here, the developer) define both the **Chisel implementation** of the computation kernel and the design space to be explored — exposed as **parameters** — using the annotation system introduced in Section IV-A, and define all the transforms — *e.g.* estimation or synthesis — to be runned in order to expose best fit at the end of the process.

[3]Thresholds are arbitrary and shall depend on estimators accuracy.



(a) Exhaustive synthesis strategy (*used as baseline*)



(b) Gradient strategy

Fig. 5: Compared exploration strategies

V. Experiments and Results

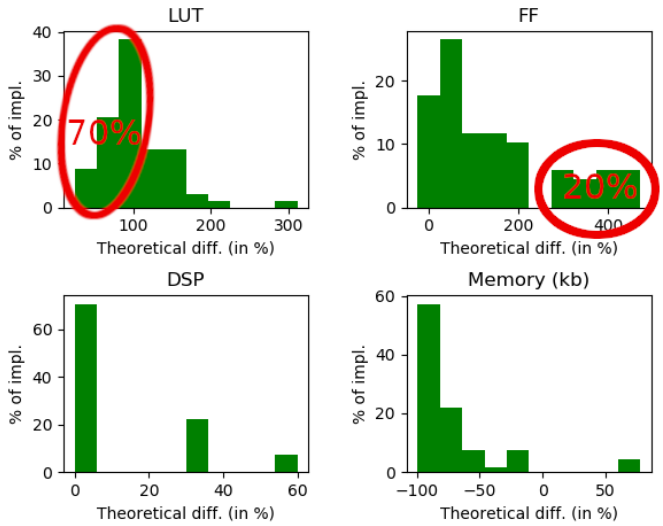
For our experiments, we targeted a **Xilinx VC709 board** using **vivado 2017.3** for synthesis, on a **12 core server** running at **3.47 GHz** with **80 GB of RAM**. We used **4 parallel threads** and define a **2 hours timeout** for synthesis processes, in order to keep memory usage under resource constraints.

A. Estimators accuracy

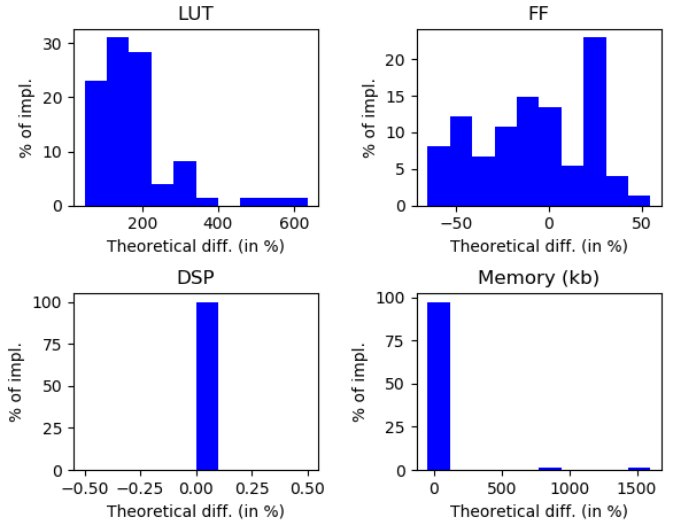
In order to quantify accuracy and fidelity of our FIRRTL based estimators, we aimed to compare resource usage estimation and synthesis results — considered as theoretical values — for the 4 considered resources metrics (LUTs, Flip Flops, DSPs and BRAMs) over **168 different GEMM implementations** — varying also **element bitwidth** over `@pow(4, 7)` in contrast to Figure 3. Among those 168 implementations, 82 were dropped, either due to incompatible parameters combinations as mentioned in Section IV-A, or to synthesis timeouts, resulting in a combination of **86 different implementations** in Figure 6. For coherent comparisons, we define relative difference between a theoretical value y and its estimated counterpart $\hat{y}$ as the result of $\frac{\hat{y}-y}{y}$.

Using 2 hours timeouts, it took **52:47:02 hours** to synthesize the whole space, while it took **01:37:59 hours** for exhaustive FIRRTL estimations without macro block replacement (Fig. 6a) and **06:37:28 hours** with macro block replacement (Fig. 6b). As macro replacement is a complex transform needing to traverse and modify FIRRTL representation of circuits, we remark it is a time-consuming operation, but results in more accurate estimations as demonstrated in this section. Thence, macro replacement process might be optimized in order to both restrain computing resource usage and increase estimation speed, and it is considered as future work.

Considering no macro block replacement, Figure 6a can be decomposed in 4 parts — one for each considered resource — where each bar of the histograms represents the percentage of implementations with similar relative differences. For example, we observe that more than 70% of implementations are overestimating LUT usage of a factor varying in $[\![0, 100\%]\!]$ of real usage. Flip Flops are over estimated of almost 400% for more than 20% of implementations — which might result in erroneous choices for exploration — and DSP usage is perfectly estimated for more than 60% of implementations. By comparing Figure 6a and 6b, we observe that using macro replacement decreases LUT estimation accuracy — while maintaining more than 60% of implementation estimated in $[\![0, +200\%]\!]$ of real usage — but increases accuracy of the three other estimators. We can also observe that, even with macro replacement, LUTs and Flip Flops are estimated with significant errors, resulting in non accurate estimations.



(a) Relative difference (in %) without macro block replacement



(b) Relative difference (in %) when using macro block replacement

Fig. 6: Accuracy of FIRRTL based estimators, with respect to synthesis results — using 86 GEMM implementations

However, we can highlight three remarkable tendencies when using macro block replacement:

- DSPs and BRAMs are almost perfectly estimated
- LUTs are always overestimated, in an interval of $[\![0, +200\%]\!]$ for more than 70% of implementations
- Flip Flops are estimated within $\pm 50\%$ of theoretical value

This way, even if FIRRTL based estimations are not accurate enough to be considered as a contribution on their own, we aim at demonstrating that they can be used for efficient exploration nonetheless.

For the next section, we use macro block replacement before resource estimation, based on observed results.

B. Design Space Exploration

As stated in Section IV-B, we aim at comparing two exploration strategies to demonstrate the usability of rapid FIRRTL estimators for exploration convergence. As shown in Section V-A, when using macro replacement, DSP amount is always estimated accurately, and LUT amount is mainly overestimated of a factor varying between 0% and 200%. Moreover, GEMM kernels are **computation intensive**, meaning that computation resources — *i.e.* LUTs and DSPs — are often critical for implementations. Thereupon, for the gradient strategy (Fig. 5b), we define the pruning criterion as done in Section IV-B — this time with resource usage considerations:

$$P(imp_i) = (\%_{DSP}(imp_i) > 100\%) \vee (\%_{LUT}(imp_i) > 200\%) \quad (2)$$

where $\%_{DSP}(imp_i)$ and $\%_{LUT}(imp_i)$ respectively represent estimated consumption of DSP and LUT for an implementation imp_i.

For this exploration process, we chose to maximize design theoretical throughput (in GOp/s), using a simple analytical formula to derive it from both generation parameters and operating frequency as given by the result of synthesis processes. As Ferres et al. [22] showed that experimental latency obtained in simulation only differed from theoretical latency by few cycles, which is negligible with respect to the order of magnitude of the problem, we use theoretical throughput as a performance metric for such exploration. Based on the same work, we used the maximum percentage of resource usage for the 4 resource considered — LUTs, Flip Flops, DSPs and BRAMs — as cost metric, as saturating one resource will result in non placeable designs.

Figure 7 shows synthesized implementations that did not timeout for both strategies, representing the evolution of theoretical throughput of designs with respect to occupied area. We can remark that the gradient based strategy requires much less synthesis processes (represented as red triangles) than the exhaustive strategies (represented as blue dots), but still considers the best design — as of the meaning of throughput — that fits on the target board (*i.e.* with $area \leq 1.0$). Indeed, all implementations synthesized during the gradient based exploration were also synthesized during the exhaustive process.

Table I compares the temporal behaviour of each strategy. Both strategies find the same best fit with an achieved throughput of 231.334 GOp/s, but the gradient strategy requires $\times 7$ less synthesis than the exhaustive strategy, resulting in a $\times 4.1$ faster convergence. This is mainly due to a reduction of synthesis timeouts, as synthesis runs which does not converge in the exhaustive strategy are estimated at RTL level as consuming too many resources, and are not even considered as candidates for synthesis in the gradient based strategy.

Strategy	Best throughput	#(space)	#synth (#timeout)	Time	Speed-up
Exhaustive	*231.334 GOp/s*	41	*41 (19)*	*13h51m56s*	-
Gradient	231.334 GOp/s		6 (1)	03h21m06s	$\times 4.1$

TABLE I: Comparing exploration strategies.

Achieved optimal implementation reaches a theoretical performance of 231.334 GOp/s on target board, which is coherent with the original solution [22] — and is comparable to state of the art solutions — showing that process did not alter design performances.

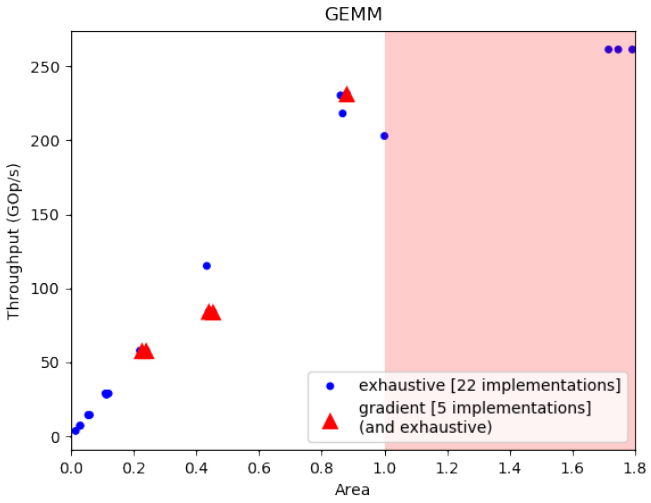


Fig. 7: Synthesized design spaces for both strategies.

VI. Conclusion

Hardware Construction Language is a novel paradigm enabling hardware reusability while maintaining control over generated designs, and HCL-based initiatives are currently being proposed in order to increase developers productivity.

In this work, we demonstrate that Intermediate Representation based resource estimators can be built using FIRRTL representation, and can be used to build *ad-hoc* strategies for efficient design space exploration. Over a GEMM use case, we achieve state of the art performances — 231.334 GOp/s for 32 wide unsigned integers — while increasing convergence speed by a $\times 4.1$ factor and limiting synthesis runs by a $\times 7$ factor.

We now aim at proposing a generic methodology for hardware generator design and exploration, as well as *proof-of-concept* framework to enable user-defined efficient design space explorations over Chisel based kernels. We claim that scala high level features can bring a lot to hardware developers productivity, and aim at leveraging such features in a comprehensive manner. We also aim at allowing users to explore on non architectural features — *e.g.* quality of result — to provide a more generic exploration framework.

References

[1] B. C. Schafer and Z. Wang, "High-Level Synthesis Design Space Exploration: Past, Present, and Future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, Oct. 2020.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, June 2012.

[3] C. Shi, J. Hwang, S. McMillan, A. Root, and V. Singh, "A System Level Resource Estimation Tool for FPGAs," in *Field Programmable Logic and Application*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.

[4] A. Nayak, M. Haldar, A. Choudhary, and P. Banerjee, "Accurate area and delay estimators for FPGAs," in *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, (Paris, France), IEEE Comput. Soc, 2002.

[5] P. Bjureus, S. Avionics, M. Millberg, and A. Jantsch, "FPGA Resource and Timing Estimation from Matlab Execution Traces," 2002.

[6] X. Gao, J. Wickerson, and G. A. Constantinides, "Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (Monterey California USA), ACM, Feb. 2016.

[7] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-Aware Throughput Optimization for High-Level Synthesis," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, (Monterey California USA), ACM, Feb. 2015.

[8] Y. L. Aung, S.-K. Lam, and T. Srikanthan, "Rapid estimation of DSPs utilization for efficient high-level synthesis," in *2015 IEEE International Conference on Digital Signal Processing (DSP)*, (Singapore, Singapore), IEEE, July 2015.

[9] R. Meeuws, S. A. Ostadzadeh, C. Galuzzi, V. M. Sima, R. Nane, and K. Bertels, "Quipu: A Statistical Model for Predicting Hardware Resources," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 6, May 2013.

[10] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators," in *Proceedings of the 53rd Annual Design Automation Conference*, (Austin Texas), ACM, June 2016.

[11] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design Space exploration of FPGA-based accelerators with multi-level parallelism," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, (Lausanne, Switzerland), IEEE, Mar. 2017.

[12] L. Nardi, D. Koeplinger, and K. Olukotun, "Practical Design Space Exploration," in *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, vol. 1, (Rennes, FR), IEEE, Oct. 2019.

[13] T. Todman and W. Luk, "Reconfigurable Design Automation by High-Level Exploration," in *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, vol. 1, (Delft, Netherlands), IEEE, July 2012.

[14] Dong Liu and B. C. Schafer, "Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, vol. 1, (Lausanne, Switzerland), IEEE, Aug. 2016.

[15] C. Schmidt and A. Izraelevitz, "A Fast Parameterized SHA3 Accelerator," vol. 1, 2015.

[16] L. Deng, K. Sobti, and C. Chakrabarti, "Accurate models for estimating area and power of FPGA implementations," in *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, (Las Vegas, NV, USA), IEEE, Mar. 2008. ISSN: 1520-6149.

[17] P. Schumacher and P. Jha, "Fast and accurate resource estimation of RTL-based designs targeting FPGAS," in *2008 International Conference on Field Programmable Logic and Applications*, (Heidelberg, Germany), IEEE, 2008.

[18] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Transactions on Mathematical Software*, vol. 5, Sept. 1979.

[19] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of FPGA-based Deep Convolutional Neural Networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, (Macao, Macao), IEEE, Jan. 2016.

[20] J. Shen, Y. Qiao, Y. Huang, M. Wen, and C. Zhang, "Towards a Multi-array Architecture for Accelerating Large-scale Matrix Multiplication on FPGAs," Mar. 2018. arXiv: 1803.03790.

[21] K. S, J. Mathew, B. R. Jose, and N. S, "UniWiG: Unified Winograd-GEMM Architecture for Accelerating CNN on FPGAs," in *2019 32nd International Conference on VLSI Design and 2019 18th International Conference on Embedded Systems (VLSID)*, (Delhi, NCR, India), IEEE, Jan. 2019.

[22] B. Ferres, O. Muller, and F. Rousseau, "Chisel Usecase: Designing General Matrix Multiply for FPGA," in *International Symposium on Applied Reconfigurable Computing*, Springer, 2020.

[23] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017.

[24] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," Tech. Rep. UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[25] Xilinx, "7 series dsp48e1 slice user guide." https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf, 2018.