A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model

Marco Siracusa, *Member, IEEE,* Emanuele Del Sozzo, *Member, IEEE,* Marco Rabozzi, Lorenzo Di Tucci, *Member, IEEE,* Samuel Williams, Donatella Sciuto, *Fellow, IEEE,* Marco Domenico Santambrogio, *Senior Member, IEEE*

Abstract—With reconfigurable fabrics delivering increasing performance over the years, Field-Programmable Gate Arrays (FPGAs) are becoming an appealing solution for next-generation High-Performance Computing (HPC) systems. However, in order to gain traction among traditional von Neumann architectures, the optimization process of FPGA designs should be further abstracted to a higher level. In fact, while High-Level Synthesis (HLS) already provides a handy way to write FPGA code with common high-level languages, substantial effort and expertise are still required to optimize the resulting FPGA design for the underlying hardware. To overcome this problem, we propose a semi-automated performance optimization methodology based on a Hierarchical Roofline model for FPGAs. System-wide and applications-specific optimizations such as off-chip memory transfer and data locality optimizations are guided by the FPGA Roofline model whereas FPGA-specific optimizations are automatically searched by a Design Space Exploration (DSE) engine. We demonstrate the way this methodology allows to easily analyze and optimize to peak system performance a wide set of applications ranging from particle methods, wavefront algorithms, and sparse arithmetic computations. In addition, we prove that the integrated DSE engine achieves a 14.36x maximum speedup if compared to previous automated solutions in the literature.

Index Terms—Roofline performance model, FPGA, High-Performance Computing, Hardware Accelerator Design

1 Introduction

The newest HPC systems integrate high-performance co-processors to efficiently execute compute-intensive tasks out of the general-purpose pipeline [1]. Among the co-processors currently on the market, FPGAs are becoming an appealing solution due to their increasing performance and power efficiency [2]. Nowadays, thanks to advanced HLS tools [3], FPGA users can write programs in common high-level languages and optimize the resulting designs by means of HLS pragmas [4]. However, while these pragmas allow to make the best usage of the underlying FPGA resources, (1) properly setting and tuning pragmas requires a substantial effort and (2) HLS pragmas cannot generally optimize off-chip memory transfer. Hence, time and expertise are still required to achieve optimal FPGA designs [5].

When a similar problem arose for CPUs, research and industry proposed several optimizations tools to assist the development flow. Among these, the Roofline model [6] is a visual method that, in a single performance figure, visualizes attainable system performance and optimization strategies in terms of computation, off-chip memory bandwidth, data locality and, in later versions, cache hierarchy [7] (after which the Hierarchical Roofline model is named). Because of its intuitiveness, this model has become a confirmed methodology to optimize HPC applications for multi-cores [7], [8] and GPUs [9], [10] and we believe it could substantially improve the FPGA optimization process too. However, this model has been originally formulated for von Neumann architectures where the peak performance (e.g. GFlops/s) could be computed empirically. As this does not hold for reconfigurable architectures [11], the original Roofline model is not directly applicable to FPGAs.

For this reason, we firstly formulate an FPGA version of the Hierarchical Roofline model to intuitively guide the design optimization of high-level aspects such as off-chip memory transfer and data locality. Then, we integrate the FPGA Roofline model with an automated DSE engine taking care of exploring pragma-based FPGA optimizations, providing a novel semi-automated FPGA optimization process for HPC HLS applications.

After introducing the Roofline model theory (Section 2) and related FPGA work (Section 3), we discuss the **novel contributions** of this work, which are:

- An analytical Hierarchical Roofline model formulation for FPGA devices (Section 4.2).
- An automated methodology to compute the FPGA Roofline components such as peak performance (Section 4.2.1), off-chip transfer (Section 4.2.2), and data locality (Section 4.2.3) starting from an HLS code.
- An integrated DSE engine (Section 4.3) to explore the optimization space given by HLS pragmas.
- A comprehensive FPGA off-chip memory transfer analysis and modeling (Section 5).

We then illustrate how to use such methodology to optimize common HPC applications and achieve peak system performance with low design effort (Section 6 and 7).

- *Marco Siracusa, Emanuele Del Sozzo, Donatella Sciuto and Marco D. Santambrogio are with the DEIB of Politecnico di Milano, Milan, Italy. E-mails: marco.siracusa@mail.polimi.it, {emanuele.delsozzo, donatella.sciuto, marco.santambrogio}@polimi.it*
- *Marco Rabozzi and Lorenzo Di Tucci are with Huxelerate s.r.l., Italy. E-mails: {marco.rabozzi, lorenzo.ditucci}@huxelerate.com*
- *Samuel Williams is with the CRD of the Lawrence Berkeley National Laboratory, Berkeley, California, USA. E-mail: swwilliams@lbl.gov*

Manuscript received April 19, 2005; revised August 26, 2015.

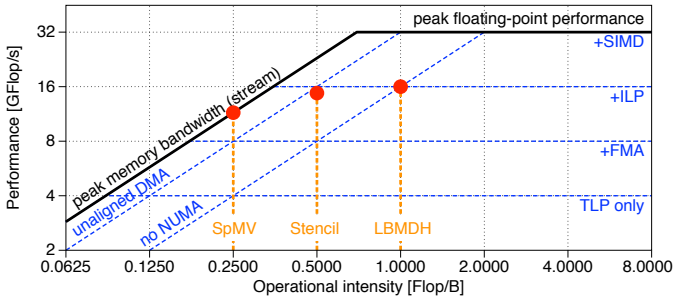


Fig. 1: *IBM QS20 Cell Blade* [12], [13], [14] Roofline model.

2 The Roofline model

The Roofline model intuitively visualizes performance bottlenecks and optimization guidance for a given architecture. As shown in Fig. 1, the Classical Roofline formulation [6] models CPU performance with respect to floating-point arithmetic and off-chip memory traffic. The maximum empirical value of these components is defined as the *peak floating-point performance* FP [GFlops/s] and the *peak DRAM bandwidth* BW [GB/s] respectively. Defining the application *operational intensity* OI [GFlops/B] as the floating-point operations performed per byte of DRAM traffic, the attainable performance can be computed as $P = min(FP, BW \times OI)$. The intersection of peak bandwidth and peak performance is called *ridge point*. Applications with OI leftmost the ridge point (SpMV [12], Stencil [13]) are called *memory bound* as limited by the DRAM memory traffic. Applications with OI rightmost the ridge point (LBMDH [14]) are called *compute bound* as limited by the floating-point arithmetic.

The first optimization step consists of calculating or profiling the operational intensity and the runtime performance of the given application. This way, the user can understand the current performance bottleneck and the room left for improvement. If the code is memory bound, the user should consider to improve the data locality of the application to move farthest right on the operational intensity axis. Once worked on the operational intensity, *memory ceilings* and *computational ceilings* should be used to define an optimization strategy. A ceiling is computed through empirical measurement of the system performance on a certain suboptimal configuration avoiding specific optimizations. Therefore, if the user wants to achieve higher performance than a specific ceiling, the associated optimizations should be applied. Drawing a vertical line in correspondence of the obtained operational intensity, the intersecting ceilings represent the required optimizations to achieve optimal performance.

During the years, the Roofline model has been extended to a Hierarchical version [9], [10] breaking down the cache usage for each level. If OP is the number of operations performed by the application (which is constant throughout the cache hierarchy), x is some level of cache, and D_x is the data transferred by the particular level of cache x, then the operational intensity OI_x [GFlops/B] of the level x is computed as $OI_x = OP/D_x$. Combining OI_x with the peak bandwidth BW_x for each x-th cache level, the Hierarchical Roofline model is computed by superposition. The cache level with performance closer to its peak is the first potential bottleneck in the cache hierarchy.

TABLE 1: FPGA optimization methodologies overview

Work	Performance metric	Peak estimation		Optimization guidance		
		Performance	Bandwidth	Area	Memory	Locality
[15]	OPS/s	Empirical	Empirical	None	None	None
[16]	OPS/s	Empirical	Empirical	None	None	None
[11]	byte-ops/s	HLS reports	Empirical	None	None	None
[17]	kernel-exec/s	Synth reports	Empirical	None	None	None
[18]	kernel latency	None	None	Analytical DSE	None	None
Our	User-defined (Sec. 4.2.1)	Analytical (Sec. 4.2.1)	Analytical (Sec. 4.2.2)	DSE (Sec. 4.3)	Hierarchy + BW ceilings + BW plots (Sec. 5.3)	Locality walls (Sec. 4.2.3)

3 Related work

One of the simplest Roofline models for FPGAs is perhaps the one proposed by Muralidharan et al. [15] where, similarly to the Classical Roofline model [6], peak performance and peak bandwidth of a board are measured empirically. J. de Fine Licht et al. [16], instead, suggest having different peaks to be chosen according to the operation balance (e.g., num_adds/num_muls) of the application. Conversely, da Silva et al. [11] propose to compute the Roofline model for a single HLS design rather than FPGA board. With this approach, starting from HLS latency and area estimations, a user should manually build an FPGA Roofline model to figure out if memory or area optimizations are needed. Besides being a quite time consuming procedure, their FPGA Roofline model is built upon a FPGA-specific performance metric (*byte-operations-per-second*) that, despite allowing a more accurate modeling of (peak) design performance, it is not suitable to characterize traditional HPC workloads and architectures [1]. Yali et al. [17], instead, use a similar approach but measure performance as kernel executions per second. Anyways, we believe that even this metric is still not general enough to easily characterize HPC workloads.

Another shortcoming of the presented works is to mainly propose basic Roofline models only composed of peak bandwidth and peak performance, providing no guidance over locality optimizations nor pragma selection. Nevertheless, when optimizing HLS codes, pragma selection in particular is a quite time consuming process. Hence, several automated solutions have been proposed to efficiently explore a large set of architectural optimizations for HLS designs [18], [19]. Among these, static-analysis model-based approaches are quite promising since offering a reduced execution time. In particular, COMBA [18] implements a guided search to explore the design space of HLS-pragma optimizations. For each explored design, it constructs a data-flow graph of the kernel function and analytically estimated performance and resource usage. These estimations, however, do not model LUT usage, which may be the most constraining resource in logic-bound designs. Moreover, COMBA does not deal with off-chip memory transfer and data locality optimizations. With memory performance lagging behind processing speed [20], off-chip bandwidth optimizations would be increasingly critical for future-generation systems.

Overall, we believe that a comprehensive optimization methodology should consider (1) off-chip memory transfer, (2) data locality, and (3) resource usage (through pragma selection). However, as shown in Tab. 1, none of the previous approaches considers these all. Hence, our methodology proposes to combine an FPGA Roofline model to guide system-wide memory optimizations and a DSE engine to explore FPGA-specific optimizations, overcoming the limi-

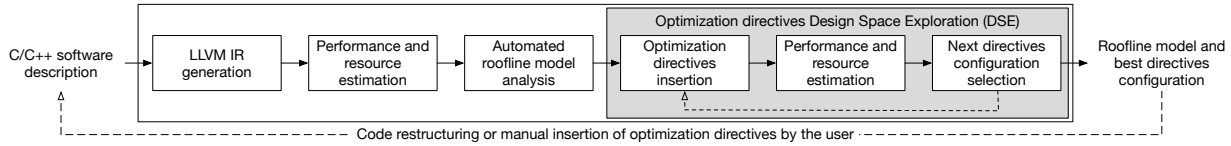


Fig. 2: Overview of the proposed methodology.

tations of these two methodologies taken singularly. In this way, the user can rely on a comprehensive methodology to iteratively optimize both memory-bound and compute-bound applications. The FPGA Roofline model we propose, conversely from previous models [11], [15], [16], [17], implements a hierarchical version that accurately estimates peak performance analytically, according to a user-defined performance metric, and integrates novel features to guide memory-transfer and data-locality optimizations. For consistency with our FPGA Roofline model, we propose a DSE engine that, conversely from previous works [18], [19], also integrates off-chip transfer and LUT usage estimations along with specific pre-processing IR-level optimizations [21], [22].

Section 4, 5, and 6 detail the implementation and usage of the proposed features whereas Section 5 highlights the value thereof. In particular, Section 7.1, 7.2, and 7.3 illustrate the way our methodology, thanks to the novel features listed in Tab. 1, allows easily achieving state-of-the-art performance on complex HPC workloads where other methodologies fall short. Instead, Section 7.4 compares our DSE engine against COMBA where, on fully-cached designs, we achieved a 14.36x maximum speedup.

4 Proposed Methodology

FPGAs-based accelerators are generally composed of an off-chip DRAM storage (such as High-Bandwidth Memory (HBM) and/or DDR SDRAM banks) and the FPGA fabric itself. This fabric is composed of a reconfigurable-logic matrix embedding specific DSP blocks and additional on-chip storage also known as Block-RAM (BRAM).

While the literature mainly focuses on automatically optimizing FPGA usage by exploiting instruction-level and thread-level parallelism of fully-cached designs [18], many scientific applications require off-chip memory storage. Therefore, we propose to guide global-memory optimizations through FPGA Roofline components whereas on-chip optimizations are automatically explored by the DSE engine. To allow an automated analysis of C/C++ HPC applications, this methodology comes as an LLVM-based analysis tool detailed in Fig. 2. The tool takes as input the target FPGA specifications and a *C/C++* algorithm enriched with optimizing directives compliant with the Vivado HLS specifications [3]. We support directives for *loop unrolling*, *loop pipelining*, *loop flattening*, *array partitioning* and *global memory mapping*. Similarly to commercial HLS tools [3], we require compile-time known loop bounds to allow static analyses.

As a first step, our tool compiles the *C/C++* input code into the LLVM Intermediate Representation (IR) then statically analyzes it to provide an initial performance and resource estimation. The performance is estimated through a scheduling simulation of the kernel function whereas the resource usage is estimated considering the resource utilization of the scheduled operators. In order to support this phase, our methodology features a profiling library that, for each tuple *(LLVM-operator, type, frequency)*, contains its performance (latency and timing) and resource usage (BRAM, DSP, FF and LUT) estimations.

As a second step, our tool automatically generates the Roofline model where performance bounds, performance ceilings, locality walls, operational intensity and estimated performance of the current design are visualized. At this point, the user can either restructure the code and overcome any memory or locality bottleneck identified by the Roofline analysis or proceed with the automated DSE of optimization directives. Once executed, the DSE plots the most promising results directly on the Roofline model. In this way, the user can exploit the Roofline analysis to individuate a suboptimal DSE exploration and use the Roofline components to investigate the causes preventing peak performance achievements. In the rest of the Section we detail performance estimation (Section 4.1), FPGA Roofline generation (Section 4.2) and DSE generation (Section 4.3) of an HLS application.

4.1 Performance And Resource Estimation

For a reduced execution time, the performance and resource usage of a given design are estimated directly on the LLVM IR. A preprocessing phase optimizes the IR with HLS-specific passes for high estimation accuracy. In detail, after running a base *-O1* recipe, we run the following custom passes: function versioning, loop unrolling, tree balancing of associative operators, propagation of constant memory accesses, advanced range analysis for types manipulation, instruction redundancy elimination and resource mapping for multiply-add operations. We then run selected built-in LLVM Passes as *loop-rotate*, *simplifycfg*, *instcombine*, *GVN* and heuristic *inline* proven beneficial for HLS [21], [22], [23].

For a given kernel, we compute its latency and resource usage by executing a scheduling simulation of each LLVM basic block and aggregating the result according to the kernel data-flow graph. In case of loops, the resulting latency depends on whether the loop is pipelined. If no pipelining is applied, the loop latency is computed as the product of the iteration latency IL and the number of iterations N. For pipelined loops, instead, the loop latency is computed as $IL+II\cdot(N-1)$ where II is the initiation interval of the loop. In order to speedup the result generation, we approximate the II as the minimum initiation interval without running time-consuming modulo scheduling algorithms [24].

The latency of a single basic block is computed with a custom *Resource-Constrained List-Based Scheduler* integrating instruction chaining and resource sharing. Similarly to other approaches [18], we assume all operations unconstrained except from memory accesses, which are regulated by partition queues. As such, all local and global memory accesses

to a single array partition are inserted in priority queues scheduling a restricted amount of accesses for each cycle (e.g. 2 concurrent accesses/cycle for BRAM or 1 concurrent access/cycle for DRAM/HBM). Each memory access is assigned to a partition queue according to its base address and access pattern (found via *llvm::ScalarEvolution*) and by the partition strategies applied to the base vector. In this phase, statistics are collected and forwarded to the Roofline generator and DSE controller described next.

4.2 Automated Roofline Model analysis

This Section discusses how we adapt the Classical Roofline model [6] to FPGAs, expand it to a hierarchical version [9], [10] and automatically compute it starting from a given algorithm and target device. The Classical Roofline model computes both the peak bandwidth and the peak performance only considering the target architecture. However, when targeting FPGA devices, it is not accurate to provide a peak performance independent from the target computation to accelerate [11]. In fact, different operations have a different resource consumption. Hence, the number and types of processing elements that can be configured in parallel, and so the peak performance, strongly depends on the operations to perform. For this reason, we propose a methodology to compute the peak performance that accounts both for the considered application and the target FPGA.

4.2.1 Peak Performance

We compute the peak performance considering an ideal hardware implementation of the algorithmic behavior then rescaled to fit the device characteristics. This ideal implementation assumes no data dependencies across operations, no resource constraints, full thread-level parallelism and full instruction-level parallelism. In this way, the ideal implementation assumes to instantiate a dedicated hardware operator for each runtime occurrence o_{op} of a certain operation *op* in the LLVM IR. The runtime occurrences o_{op} are computed through static code analysis considering the scope of the operation *op*, the loop nest it resides in and the function context in the callgraph.

However, there are some spurious LLVM operations that should not be counted when computing the peak performance. For example, the LLVM IR contains operators implementing control logic needed by loops to increment and compare the induction variables at each iteration. If this logic implements a behavior known at compile time, it is completely constant-propagated when loops are fully unrolled, introducing inaccuracy in the peak performance computation. Thus, we compute the peak performance considering the restricted set OP of "algorithmically-useful" unroll-invariant operators in the LLVM IR.

For each operation (e.g., a floating-point sum), we associate a corresponding operator (e.g., a floating-point adder) that is physically implemented on the FPGA. We consider the operators as fully pipelined and working at the target clock frequency f. Hence, at each clock cycle, an operator can produce a valid result. Ideally, an implementation with maximum performance on FPGA would require to instantiate each operator as many times as the corresponding operations count o_{op}. Such implementation requires c_r resources for each resource type $r \in R = \{BRAM, DSP, FF, LUT\}$:

$$c_r = \sum_{op \in OP} o_{op} \times u_{op,r} \tag{1}$$

where $u_{op,r}$ is the amount of resources of type $r \in R$ required by the operator $op \in OP$, which is provided by the profiling library. Most likely, the required resources would largely exceed the FPGA availability so we rescale the ideal hardware implementation to a feasible one. We compute a scale factor SF to take into account the amount of resource reduction needed to fit within the target FPGA:

$$SF = \left\lfloor \max_{r \in R} \frac{c_r}{av_r} \right\rfloor \tag{2}$$

where av_r is the amount of resources of type $r \in R$ available on the FPGA. The actual number of operators i_{op} of type $op \in OP$ implemented in the design is scaled by a factor SF compared to the ideal implementation:

$$i_{op} = \frac{o_{op}}{SF} \tag{3}$$

and the peak performance bound (in terms of total number of operations per second) is computed as:

$$P = \sum_{op \in OP} i_{op} \times f \tag{4}$$

This formula considers that each operator can produce a useful result every clock cycle (pipelined operators) and that the operators are never idle (no data dependencies). Finally, we empirically consider as peak performance the 80% of P due to routing congestion and timing closure issues [25].

As the total number of operations per second might not provide an accurate performance indicator for HPC workloads, we allow to select a custom performance metric specifying a restricted set of operators OP_{res} to be considered for performance counting. For example, the user can express the result in terms of GFlops/s just restricting the set of operators to the floating-point entities. We additionally allow to specify the performance in terms of *generated results per second* by inserting a special directive in the scope of the source code where the results are generated. This directive is processed as a mock instruction with null resource usage and o_{res} runtime occurrences that, replacing $OP = \{o_{res}\}$ in Eq. (3) and (4), provides a performance of

$$P = \frac{o_{res}}{SF} \times f. \tag{5}$$

4.2.2 Off-chip memory bandwidth

Modern FPGA-based architectures integrate both DDR and HBM memories accessible via AXI-based controllers as detailed in Section 5. For each argument of the kernel function, the developer can create an AXI-based memory interface connected to a single DDR or HBM bank. Although this hybrid solution allows to combine the high capacity of DDR and high bandwidth of HBM, their different responses require a more complex analysis and optimization strategy.

For memory-bound applications, breaking down the off-chip transfer to each memory bank would enable a more intuitive analysis. However, since complex applications can have a large number of arguments, allowing a selective

breakdown is essential to keep the analysis clear. Thus, we firstly visualize the whole aggregate bandwidth and its related operational intensity computed with respect to the whole off-chip data transfer. Then, we allow a selective breakdown to analyze bundles of banks independently. Supposing we are analyzing an application with $A = \{arg_0, arg_1, ...\}$ arguments. For each memory bank $bank_i$ serving a set of arguments A_i, we characterize its off-chip memory transfer via analytical models (Section 5). These memory models abstract the underlying memory subsystem through a set of parameters that can be extracted from the design configuration and source code.

We define the peak bandwidth as the maximum sustainable bandwidth of the bank. However, there are cases where this bare information is not informative enough. In fact, in case a bank does not reach the peak bandwidth, it is unclear whether it is caused by a suboptimal memory configuration or a poor memory access pattern. For this reason, we introduce the *peak configuration bandwidth* and memory ceilings. The peak configuration bandwidth (Section 5.1) represents the attainable peak bandwidth with respect to the interface bitwidth and design frequency. In case of suboptimal configuration, the developer should consider, for example, to implement optimizations such as memory access coalescing or data reshaping to maximize memory port usage and increase attainable performance.

The *memory ceilings* (Section 5.2), instead, represent the achievable bandwidth of common access patterns such as random access or data-dependent access. The user can exploit these ceilings as a performance target to eventually understand if the system needs further interface tuning. For example, increasing the FIFO size storing the requests of an argument performing random access leads to peak bandwidth but it comes with certain area requirements to be considered. In the same way, increasing the memory concurrency of a bank improves data-dependent access performance but it introduces a banking problem to be considered. Looking at these ceilings, the user can assess whether the tuning would bring to actual bandwidth improvements or it would be overshadowed by other major bottlenecks. If the user wants to visualize a set of banks as an aggregate entity, the single Roofline components are summed up.

4.2.3 Operational intensity and locality walls

The operational intensity measures the work done per off-chip transferred byte. In our model, the work done is expressed by Eq. (5) whereas the off-chip transfer is computed per bank as just illustrated. Therefore, the operational intensity of a certain $bank_i$ is

$$OI_{bank_i} = \sum_{op \in OP_{res}} \frac{o_{op}}{tb_{bank_i}} \tag{6}$$

where tb_{bank_i} is the total amount of bytes accessed for bank i. If the resulting operational intensity is leftmost the ridge point, the user may consider to optimize the data locality of the considered channel. In case the user wants to aggregate different banks in a group G, the aggregate operational intensity would be computed as

$$OI_G = \frac{\sum_{op \in OP_{res}} o_{op}}{\sum_{bank_i \in G} tb_{bank_i}} \tag{7}$$

```
def run_dse(kernel_ir)
  opt_stack = <empty>
  loop_info = initialize_loop_info(kernel_ir)
  opt_step = "loop"
  do
    iterate = False
    opt_ir = insert_opt_directives(kernel_ir, opt_stack)
    stats = analyze_ir(opt_ir)
    if stats.design_area > device_area
      (last_opt_loop, prev_opt) = opt_stack.pop()
      loop_info[last_opt_loop].opt = prev_opt
      loop_info[last_opt_loop].fully_optimized = True
      iterate = True
    else
      if opt_step == "loop"
        loop = select_loop_to_optimize(loop_info, stats)
        if loop != None
          opt_stack.push((loop, loop_info[loop].opt))
          loop_info[loop].opt = select_optimization(loop)
          iterate = True
          opt_step = "array"
        else
          last_loop = opt_stack.get_last()
          foreach array in last_loop.arrays
            array.opt = get_partitioning(array, stats)
          iterate = True
          opt_step = "loop"
  while(iterate)
  return extract_optimizations(opt_stack)
```

Algorithm 1: Design Space Exploration algorithm

Whereas multicore and GPU integrate cache managers to automatically reduce off-chip memory traffic, FPGAs leave the data locality optimization to the user. In order to visualize the locality requirement of each loop and provide an optimization guidance, we introduce the *locality walls*. These walls are vertical lines on the Roofline model representing the impact on operational intensity of different cache strategies applied to the considered argument arg_i.

The operational intensity of a configuration is estimated simulating the effect of a scratchpad cache placed at a certain hierarchy of the loop nest and caching all the accesses to arg_i performed by inner loops. In particular, the total amount of bytes accessed by a loop and the cache size needed to store the temporary data is statically estimated combining *llvm::ScalarEvolution* and *llvm::LoopInfo* analysis results. In this way, the user has a clear visualization on which strategy is required to move the operational intensity into the compute bound area.

4.3 Design Space Exploration

The design space given by combining HLS optimizations of loops or arrays is generally too large to be exhaustively pictured on the FPGA Roofline. Therefore, our approach automatically explores these designs with a DSE engine and reports on the Roofline only the most promising ones.

As shown in Alg. 1, the knobs are explored iteratively alternating loop and array partitioning optimizations. Each iteration (1) inserts the optimization directives in the IR to be analyzed (`insert_opt_directives` in Alg. 1), (2) estimates latency and area of the current knob (`analyze_ir` in Alg. 1), and (3) selects the optimization to apply next (`select_optimization` in Alg. 1). The choice of the next knob to investigate (`select_optimization` in Alg. 1) is performed by a controller that alternates loop optimizations (unrolling, pipelining and flattening) and local-memory optimizations (array partitioning).

For the loop optimizations, at each step, we recursively visit the loop with higher latency contained in the function body or loop nest until an innermost loop l is reached (`select_loop_to_optimize` in Alg. 1). If loop l is not already pipelined, we attempt to pipeline it first since pipelining a loop does not increase resource usage as much as the unrolling of the same, keeping the configuration feasible and faster to be analyzed. However, if l is already pipelined, the DSE tries to achieve higher performance by further unrolling the loop. Every subsequent unrolling optimization of a loop l is done by selecting the next higher unrolling factor that divides the loop trip count. The explored knobs are saved in a stack (`opt_stack` in Alg. 1), so that, if a given knob is not feasible due to resource constraints, the corresponding loop that led to exceed the available resources is marked as *fully optimized* and is not considered for subsequent optimizations. Then, the exploration backtracks to the previous feasible knob and continues the search.

After each loop optimization, a local memory optimization follows. Indeed, after optimizing a loop, performance may be constrained by the number of available BRAM ports. Within this step, for each array a in the function and for each dimension d of array a, we collect the number k of distinct offsets used to access dimension d. Then, we partition dimension d by a factor k. The selection (`get_partitioning` in Alg. 1) among cyclic and block partitioning is done by simulating, at schedule-time, the number of conflicts that occur within the partitions. The partitioning type that minimizes the maximum number of conflicts per partition is selected. Finally, the exploration continues with a loop optimization step and terminates when all the loops are completely unrolled or marked as *fully optimized*.

5 Modeling Roofline memory components

This Section illustrates the memory models we used for computing the peak (configuration) bandwidth and bandwidth ceilings of the proposed FPGA Roofline model.

To comply with the OpenCL standard, flagship HPC boards of main FPGA vendors handle global communication through *memory buffer*s. A memory buffer is an array of fixed size that can be allocated in any global memory bank and managed by the host code through specific directives. An IP core can access these buffers via its kernel-function arguments by specifying the required interconnections at design time. The HLS tool then instantiates the logic to interconnect each memory port of the kernel to the memory bank where the assigned buffer resides into. Formally, an AXI module is called *master* if attached to a kernel-function port or *slave* if attached to the memory side.

5.1 Peak configuration bandwidth

FPGAs communicate with the global memory asynchronously. A design reaches peak bandwidth when it generates a traffic that saturates the sustainable bandwidth of each bank. To model this behavior, we firstly define the *memory quanta* Q of an AXI master interface as the amount of bytes that port is set to access each clock cycle. Now, if the kernel is running at a user-defined frequency f, the bandwidth required by that Q-wide port would be

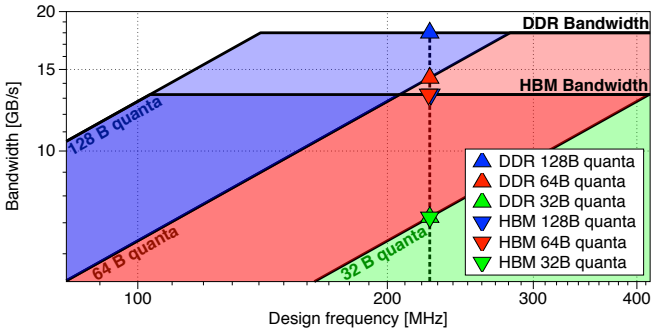


Fig. 3: Pictorial bandwidth breakdown of Eq. (8) to select optimal system configuration (design frequency and quanta) for different memory technologies (DDR and HBM).

$f \times Q$. However, defining BW_{bank} as the empirical peak bandwidth of the bank servicing the interface and W its physical port byte-width, the actual interface bandwidth is

$$BW_{interface} = min(f \times Q, BW_{bank} \times min(1, Q/W)) \quad (8)$$

which is reported on the Roofline model as the peak configuration bandwidth.

In essence, appropriately choosing f and Q is essential to avoid suboptimal bank utilization ($f \times Q < BW_{bank}$) or kernel stalls ($f \times Q > BW_{bank}$). However, if the system has several interfaces with different quantas and memories (DDR, HBM, ...), finding the optimal configuration may be difficult. Therefore, we additionally provide an intuitive performance breakdown of the achievable bandwidth that, as shown in Fig. 3, includes:

- the frequency f on the x-axis,
- the achievable bandwidth on the y-axis,
- the peak bank bandwidth of different technologies with different quantas as horizontal lines,
- the different memory quanta Q as slanted lines.

In this way, the user can easily figure out the best combination of design frequency and data coalescing to saturate the bandwidth of each bank. In fact, the plot superimposes the DDR and HBM models with different memory quantas. Tracking a vertical line on the value of the target design frequency, the user identifies the performance of each interface with respect to their memory quanta and the "frequency slack" for this particular design.

In the example in Fig. 3 targeting 225 MHz, the triangles indicate the different DDR (upward triangles) and HBM (downward) interfaces with different memory quanta (blue for 128B, red for 64B, green for 32B). A first consideration would be that the green triangles (32B) achieve same bandwidth as both limited by the too narrow quanta. In this case, for example, the user may consider to coalesce accesses and obtain the performance indicated by the red triangles (64B). Note that the DDR performance is still limited by the quanta whereas the HBM bandwidth is limited by the peak itself (since lower than DDR). Therefore, the user may consider that, for the given frequency, a 128B quanta might be used for the DDR interface whereas a HBM should use a 64B interface since increasing to 128B would not benefit.

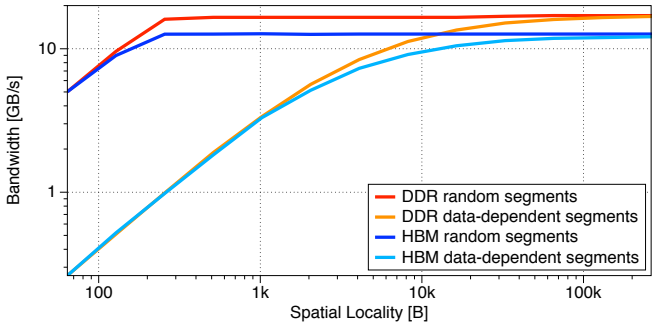


Fig. 4: HBM and DDR single-bank bandwidth accessing random or data-dependent segments and varying spatial locality. Quanta fixed to 64B and design frequency to 450MHz.

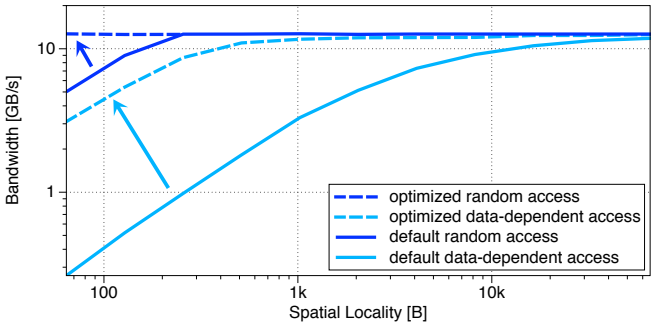


Fig. 5: Impact of increasing outstanding requests (for random access) or memory concurrency (for data-dependent access) of an HBM bank. $Q = 64B$ and $f = 450MHz$.

5.2 Bandwidth ceilings

Once the most promising design frequency and coalescing factors have been selected, the global memory access patterns should be optimized to achieve peak (configuration) bandwidth. Since this goal may sometimes be difficult (e.g., for irregular sparse computation, etc.), the proposed FPGA Roofline model provides a set of bandwidth ceilings to visualize the transfer efficiency of common access patterns. In principle, the AXI protocol performs data movements by handshaking (sends a channel request, transfers the data, and closes the channel). For compile-time unpredictable access patterns, this handshake has to be done for each memory access. Predictable stream access patterns, instead, can handshake once-per-stream instead of once-per-access. In this setting, formally known as *burst mode*, a binding handshake prior to a loop allows the core to send or receive a new data of the stream (a *beat*) any loop iteration without the need of continuously synchronizing with the memory.

In order to quantify the impact of the channel requests overhead, we implemented a microbenchmark testing different access patterns. In particular, we firstly evaluate the response of the memory subsystem over two dimensions like spatial locality and memory concurrency. The *spatial locality* is the number of bytes accessed by a certain burst. It can be calculated multiplying the memory quanta for the *burst length*. Finally, the *memory concurrency* is the amount of outstanding transactions in the memory subsystem. For the sake of clarity, we refer to the subsequent locations accessed within a burst transfer as *locality segment*.

We classify as *random access* any pattern in which the access location of a locality segment is not dependent from the previous segments. We classify as *data-dependent access* any pattern in which the access location of a certain locality segment depends on the previous segments. In both cases, we spawn a request for each segment. However, in the random access pattern, beats and requests can be overlapped whereas they cannot for the data-dependent access pattern, exposing the controller latency for each segment.

Fig. 4 visualizes the response of HBM and DDR with respect to these two access patterns while varying spatial locality. Overall, the random access follows a *min* function over the bank bandwidth and a requests-per-second bound (slanted line). The data-dependent access, instead, follows an $\alpha - \beta$ model over the bank access-time and bandwidth.

To formalize these concepts, we analytically model the transfer behaviors. Therefore, we assume to transfer a payload of B bytes that, according to the chosen spatial locality SL, would require R requests to complete. As requests and beats of the random-access pattern can overlap, the required time would be the maximum between the time required for servicing all the requests and the time required for transferring the payload. Therefore, defining P_{BW} as the maximum payload bytes transferred per second and P_R as the maximum requests per second, the required time would be $max(\frac{B}{P_{BW}}, \frac{R}{P_R})$ and the random access patterns with compile-time predictable addresses would have bandwidth

$$BW_{rnd} = \frac{B}{max(\frac{B}{P_{BW}}, \frac{R}{P_R})} = min(P_{BW}, SL \times P_R) \quad (9)$$

Conversely, for data-dependent access patterns with compile-time unpredictable addresses we have bandwidth

$$BW_{dep} = \frac{B}{\frac{B}{P_{BW}} + \frac{R}{P_R}} = \frac{1}{P_{BW}^{-1} + SL^{-1} \times P_R^{-1}} \quad (10)$$

as channel request and payload transfer are executed sequentially over the segments. Evaluating these models while fixing the memory quanta to the port width gives the random an data-dependent ceilings shown in the Roofline. If approaching these ceilings, the user should optimize the interface (as explained next) to reach peak bandwidth.

5.3 Access patterns optimizations

As demonstrated in Section 5.2, Eq. (9) shows the way the number of outstanding requests limits the random-access performance. However, as shown in Fig. 5, some tuning mitigates this bottleneck. In fact, these outstanding requests are stored in FIFOs whose capacity can be increased up to achieving peak bandwidth in case the interface is hitting the random ceiling. To ease the optimization flow, the tool suggests an optimal queue size estimated using Little's law.

Eq. (10), instead, shows that the data-dependent access pattern is limited by the memory access time, which, however, is a technological constraint. Anyway, as shown in Fig. 5, the user can still mitigate this bottleneck introducing memory concurrency on the same bank. In practice, this is implemented by allocating several buffers on the same

bank and increasing memory concurrency. If C concurrent streams are accessing one bank, we can modify Eq. (10) as

$$BW_{concurr.} = \frac{1}{P_{BW}^{-1} + SL^{-1} \times (P_R \times C)^{-1}} \quad (11)$$

Reversing the formula gets the optimal concurrency value.

6 FPGA Roofline generation and usage

While Section 4 overviews the combined usage of Roofline model and DSE to optimize FPGA designs, this Section details the usage of each Roofline component for analysis and optimization purposes. We recall that our methodology requires, for a specific FPGA, (1) a profiling library containing the performance and resource usage of each LLVM operator and, for a specific board, (2) the results of the memory microbenchmarks measuring bandwidth for different memory technologies (Section 5), design frequency, access patterns, memory quanta and spatial locality.

Instead of building a prohibitive table storing the combination of bandwidth measurements, these are modeled via the memory curves described in Eq. (9)-(10) where only few parameters (as P_{BW} and P_R) need to be stored instead of the whole spatial-locality series. Since quite costly, all of these procedures are automatically executed offline. Given board specifications and user-defined design frequency, our methodology analyzes the input LLVM IR to compute the peak performance through Eq. (5). Computing the peak bandwidth from Eq. (9) as $BW_{rnd}(SL \to \infty) = P_{BW}$ and the operational intensity through LLVM analysis, we build the basic FPGA Roofline model delineating the attainable performance of the given design. Furthermore, we analyze the LLVM IR to extract the memory quanta of each interface that, combined with the design frequency, prints the peak configuration bandwidth of each memory port (from Eq. (8)) or an aggregation thereof. For values substantially lower the peak performance, the user should improve the bandwidth usage (by tuning quanta and frequency) according to Fig. 3. Through IR analysis, we extract the locality walls needed to guide the data-locality optimization of memory-bound designs via scratchpad caching. Finally, we use the memory curves in Eq. (9)-(10) to generate the random-access ($BW_{rnd}(SL = Q)$) and data-dependent ($BW_{dep}(SL = Q)$) ceilings given quanta and frequency. If approaching these ceilings, the user should consider the optimizations mentioned in Section 5.3 (e.g., enabling more outstanding requests or memory concurrency). For a more insightful analysis, spatial locality is extracted through LLVM analysis and reported in plots like Fig. 4. At this point, as detailed in Section 4 and illustrated in Section 7, the user may run the DSE to further optimize on-chip performance.

7 Experimental evaluation

We validated the whole methodology on several HPC kernels and benchmarks. In particular, we illustrate the optimization of the N-body physics simulation algorithm where, in few steps, we converge to peak performance. Then, starting from a memory-bound state-of-the-art Smith-Waterman implementation, we use the FPGA Roofline model to port the design to a more suitable new-generation board. Moreover, we use the FPGA Roofline model to optimize the SpMV kernel to approach peak DDR bandwidth. Finally, we evaluate the DSE efficiency on the Polybench test suite outperforming state-of-the-art approaches up to a 14.36x.

7.1 N-body simulation test case

The N-body process [26] simulates the evolution of a particle system under the influence of physical forces approximated by an all-pairs approach. Due to the high memory requirement, a plain software implementation [27] is generally memory-bound on most FPGA systems. However, we use the proposed FPGA Roofline model to easily design a compute-bound implementation and the DSE engine to hit peak performance of the target DDR-based Xilinx Virtex UltraScale+ VU9P board on Amazon Web Service (AWS).

Before starting the optimization process, we set a pairs/s (particle-pair interactions per second) performance metric by placing a result-generation directive in the innermost loop. By means of our tool, we now optimize the plain version [27] of the algorithm reporting all the steps in Fig. 6. The plotted Roofline analysis clearly visualizes that the baseline design (downward red triangle) is memory bound. Since this prevents achieving peak performance, we need to optimize data locality to cross the ridge point.

The tool assists this operation by means of the locality walls (green vertical lines). Each line indicates the impact on operational intensity if all the inner loops below a specific point would cache their accesses. Those walls indicate that fully caching the data movement would turn the design compute bound. Moreover, observing that the estimated cache size associated with the full cache wall fits the BRAM constraints, we move on modifying the kernel. We add an initial phase for copying the data into the local memories before the computation and a final phase for copying the result back in global memory after the computation.

Now that we moved the design in the compute-bound area, we can improve performance by exploiting the application parallelism. Our tool assists this process with an automatic DSE that searches architectural optimizations. However, a first DSE run returns an optimization configuration that, if compared with the peak performance, is clearly suboptimal. Before executing the DSE process, our tool performs a preliminary performance analysis that identifies possible bottlenecks. A synthetic report indicates that the loop-carried dependency on the inner loop may limit DSE performance. Since these limitations are commonly overcome by loop inversion, we rewrite the code and perform another DSE run.

Free of dependencies, the optimal design performs a tiled computation by unrolling the internal loop by a factor of 96 and then pipelining it, cyclically partitioning the local memory accordingly. This final design (upward red triangle) approaches the theoretical peak performance (just 3.7% lower) and, when executed on the AWS platform, achieves performance comparable to a bespoke state-of-the-art implementation [26]. Therefore, in the end, our methodology (1) identified the memory bottleneck of the initial design, (2) proposed a data-locality optimization to move in the compute-bound area, (3) identified the loop-carry dependency limitation and (4) provided a pragma configuration to achieve peak performance.

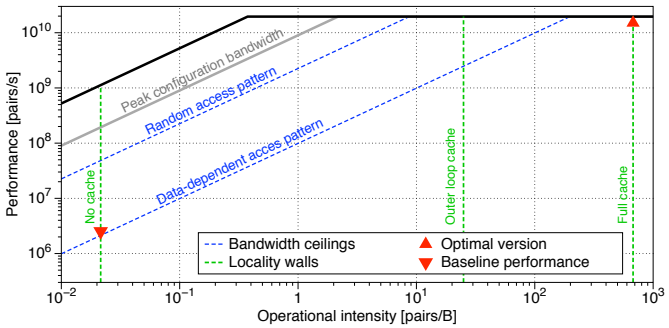


Fig. 6: N-body Roofline analysis and optimization.

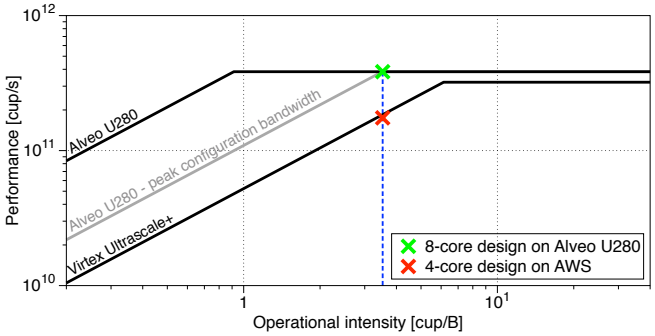


Fig. 7: Smith-Waterman analysis for multicore designs.

7.2 Smith-Waterman

Smith-Waterman [28] is a sequence alignment algorithm for identifying relationships between strings of genetic data. Given a database of length M and a query of length N, Smith-Waterman works by finding regions of similarity between subsequences of all possible lengths and holding the similarity scores in a $M \times N$ matrix. The state-of-the-art design [29] maps the characteristic *wavefront* computation of this algorithm in a systolic array that is limited by the DDR bandwidth of the target board. For this reason, we investigate how porting this design to a higher-bandwidth device would provide some performance benefit.

The first candidate platforms is the Amazon EC2 F1 instance featuring a 16 nm Xilinx Virtex Ultrascale+ FPGA with four DDR4 banks. The second option, is the Xilinx Alveo U280 integrating 32 HBM banks. Since these architectures have different bandwidth and compute capability, we rely on the Roofline model to shed light on the best choice. Generating and combining the two models, we can directly compare attainable performance and required optimization strategies. Selecting a CUP/s (cell updates per second) performance metric, the same results can also be directly compared with the literature [29], [30].

In order to maximize the port usage, we coalesce the data access to a 64B quanta and select a frequency of 280 MHz for AWS and 225 MHz for the Alveo (Section 5.1). Fig. 7 reports the superimposed Roofline models visualizing the performance bounds (black solid lines) and operational intensity (blue dotted line) of the application for the considered architectures. According to Fig. 7, the algorithm is memory bound on the AWS platform and compute bound on the Alveo U280 so different optimization strategies might be required to achieve optimal performance. In principle, the Roofline model indicates that, disregarding the adopted parallelization strategy, we can achieve a 4x or 8x performance speedup depending on the considered architecture.

A simple yet effective optimization strategy consists of increasing the task-level parallelism by replicating the main compute unit to manage different couples of query-targets in parallel. In this case, however, the memory and compute bound should be handled differently. In the memory-bound implementation, the maximum degree of parallelism is obtained saturating the memory subsystem. Performing an initial loading of query and database on the device, we can use a single 64B-wide memory port per compute unit and bind one compute unit to each of the four memory banks. In this way, we obtain the 4-core implementation whose measured performance of 174.8 GCUP/s, out of the 184.6 predicted by the model, is reported as a red cross in Fig. 7. We notice that the observed 5.3% performance loss is directly proportional to the frequency rescaling performed by the synthesis tool.

Similarly, we replicate the compute unit to approach the compute bound given by the Alveo U280. Since, as indicated by the tool, the final design would utilize a large percentage of LUTs, we expect some frequency degradation due to the final design density. Thus, we plan ahead improving the compute connectivity distributing the cores among the Super Logic Regions and HBM stacks. Moreover, since our tool suggests an 8-core implementation and the HBM provides 32 memory ports, we consider using two memory ports per core to stream input/output data at each clock cycle instead of using spare BRAMs for the in-core caching required with single port. In this way, we achieve a design with 384.2 measured GCUP/S indicated with a green cross in Fig. 7.

In the end, our methodology provided an intuitive bottleneck analysis to guide the design optimization for two different FPGA boards, overall yielding a 6.5x, 8x, and 8.5x speedup over similar CPU [31], FPGA [29], and GPU [30], implementations respectively.

7.3 Sparse matrix-vector multiplication

The sparse matrix-vector multiplication (SpMV) kernel [12] performs the multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ of a sparse matrix $\mathbf{A}$ and a dense vector $\mathbf{x}$. We encode the sparse matrix in CSR format as one of the most used [32], [33]. As such, we store the non-zero (nnz) elements of the matrix into an array `A`, where `A[i].data` and `A[i].idx` are the value and column index of the `i`-th nnz-element in row-major order. The `ptr` array, instead, stores the cumulative number of non-zero elements for each row. In this example, we use the FPGA Roofline model to analyze and optimize a memory-bound design on the Xilinx Alveo U280 when, to maximize capacity, we store `A` in a DRAM bank (2 billion fixed-point non-zeros per bank) and `ptr` in an HBM bank (64 million integer row-pointers per bank). If `x` and `y` are the input and output dense vectors respectively (allocated in HBM), the hot loops of a basic implementation are:

```
L1:for(int i=0; i<ROWS; i++)
  L2:for(int j=ptr[i]; j<ptr[i+1]; j++)
    y[i] += A[j].data * x[A[j].idx];
```

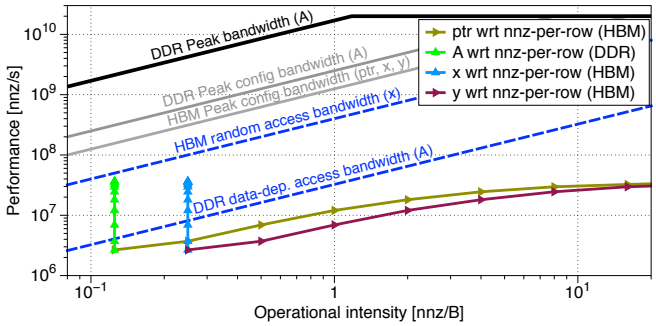


Fig. 8: Hierarchical Roofline model of the baseline SpMV design running with increasing nnz-per-row matrices.

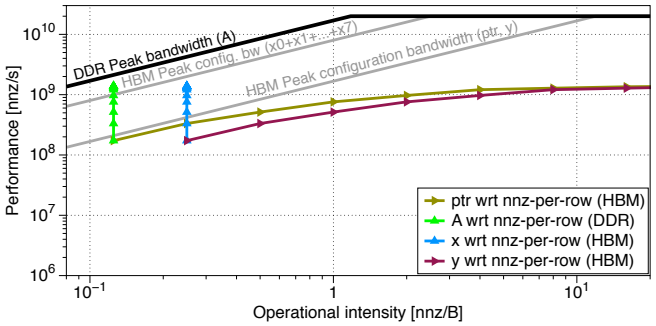


Fig. 9: Hierarchical Roofline model of the optimized SpMV design running with increasing nnz-per-row matrices.

Fig. 8 visualizes the Hierarchical Roofline model of the baseline SpMV design processing different matrices with exponentially larger $1 \leq nnz\text{-}per\text{-}row \leq 4096$ values. Considering the operational intensity of each input variable, `A` and `x` are the first potential performance bottlenecks since limited by their peak configuration bandwidth. In detail, the peak configuration bandwidth of `A` and `x` indicates that the current memory configuration could only achieve up to 12.5% of the bank bandwidth. Moreover, the actual `A` and `x` performance nearby the data-dependent ceiling indicates that memory latency is substantially impacting our design performance (only 5% bandwidth usage).

To improve the memory configuration, we coalesce the access on `A` computing multiple products in parallel over several lanes. Using the schema of Fig. 3, we can choose a suitable coalescing factor (i.e. 8) across the interfaces and a proper design frequency (i.e. 225 MHz). To avoid banks conflicts while accessing the sparse vector, each lane allocates a private copy of $\mathbf{x}$ in a dedicate HBM bank. Moreover, since $\mathbf{x}$ is accessed randomly, we should consider increasing the outstanding-request FIFOs (Fig. 5) on the HBM banks to avoid getting stuck in the blue ceiling in Fig. 8. In order to improve the access pattern, we hide memory latency using different dataflow stages.

Fig. 9 reports the performance of the optimized design achieving a speedup over the baseline design ranging from 38x to 65x. The figure indicates that, despite `A` approaches peak bandwidth for high-degree matrices ($nnz\text{-}per\text{-}row > 16$), lower-degree matrices run up to 10 times slower (due to some row-startup overhead). In case we were considering particularly low-degree workloads, we could saturate the banks' bandwidth by enforcing more memory concurrency (Fig. 5). In practice, this is done partitioning the workload over different cores all attached to the same banks. At this point, plotting the Roofline model with aggregate interfaces, we can notice that we are using a fraction of the board potential ($\frac{1}{2}$ DDR and $\frac{10}{32}$ HBM banks). Therefore, we can double the number of instantiated cores achieving a total 130x speedup over the baseline.

In the end, our methodology enabled a hierarchical bottleneck analysis that (1) highlighted a suboptimal memory interface configuration (via peak configuration bandwidth), (2) enabled a optimal coalescing-frequency choice (via Fig. 3), (3) highlighted access-pattern bottlenecks (via bandwidth ceilings), (4) visualized suboptimal performance for highly-sparse workloads, (5) guided a different optimization strategy for such case (using memory concurrency) and (6) highlighted the need of a multicore implementation to saturate bandwidth. Overall, for $nnz\text{-}per\text{-}row > 8$, our final design outperforms state-of-the-art DDR-based FPGA implementations [34] up to 1.6x, also relaxing the constraints on the dense vector size.

TABLE 2: DSE summary on PolyBench test suite.

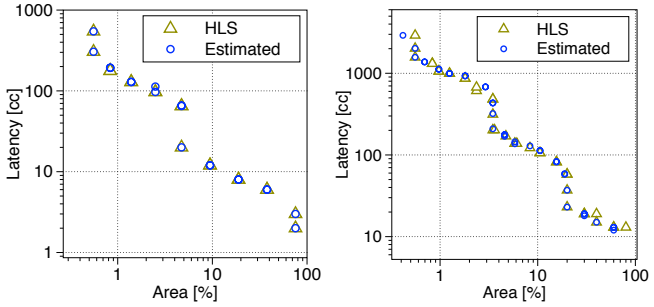
Benchmark	DSE time [s]		Design space		Speedup		Area usage [%]	
	Ours	HLS	Exaustive	Explored	wrt HLS	wrt [18]	Ours	[18]
ATAX	12.39	348.97	3.28e+7	28	664.50x	10.50x	71.11	6.67
BICG	51.89	407.79	1.44e+8	20	396.13x	4.50x	71.11	11.11
GEMM	33.21	485.76	2.62e+9	24	1090.75x	11.67x	71.11	71.11
GESUMMV	21.68	335.40	2.1e+8	19	272.50x	10.00x	75.56	4.72
MM	23.84	658.8	1.59e+13	41	364.25x	2.38x	80.00	40.00
MVT	25.65	525.59	2.62e+9	33	140.25x	2.14x	35.55	53.33
SYR2K	56.22	857.06	2.62e+9	22	688.90x	14.36x	82.22	45.94
SYRK	48.24	737.63	4.1e+6	24	667.25x	4.70x	40.00	26.67

7.4 PolyBench test suite

We evaluate the DSE accuracy, execution time, and effectiveness on the PolyBench test suite [35] and compare our results against COMBA [18]. To reproduce their settings [18], we target a Xilinx Virtex-7 device running at 100 MHz. Since COMBA assumes the design fully cached, the Roofline model intervention is not required as all the considered implementations are compute bound.

Tab. 2 summarizes the results achieved by the DSE. For each benchmark (in Column 1), we report the execution time of our DSE either using our analysis module (Section 4.1) (Column 2) or Vivado HLS (Column 3) to estimate the quality of each explored design (see also Fig. 10). Computing the design performance and resource usage directly at the IR-level and analytically modeling compute-intensive parts of the scheduling allow for an 8x to 28x saving in execution time. Column 4 reports the size of the solution space (possible combinations of optimization directives) whereas Column 5 reports the actual number of designs explored by our DSE before converging to the final solution. We also report the performance speedup (on kernel latency) achieved on the baseline unoptimized HLS implementation (Column 6) and COMBA (Column 7). Our methodology returns designs with performance up to 14.36x higher.

The main reason for the improved results comes from the additional code transformations [21], [22] applied before the estimation phase. Since both COMBA and our DSE do not explore optimizations that lead to exceeding the available



(a) Gesummv test case. (b) Mm test case.

Fig. 10: Pareto walks of DSE on two PolyBench test cases.

resources, our accurate predictions prevent stopping the search too early, driving the exploration toward solutions using, on average, 65.83% of the device area, 2.03x more than COMBA (as shown in Column 8 and 9). We provide a pictorial example through Fig. 10 that reports the DSE process of *Gesummv* and *Mm* showing latency and area usage of each explored design when estimated by our analysis module or Vivado HLS. Overall, *Gesummv* has a prediction error of 1.90% in performance and 1.24% in resource usage whereas *Mm* has a prediction error of 4.08% in performance and 5.60% in resource usage. For each plot, the projection obtained by interpolating all the designs shows how the DSE controller alternates loop pipelining (steep traits) and loop unrolling (linear traits) and converges to a 75.56% and 80% of area utilization respectively. Despite not being reported, the BRAM usage estimation is equivalent to COMBA as we use a similar model.

8 Conclusions

We presented a semi-automated methodology to optimize HPC applications for FPGA. Our novel approach combines an FPGA Roofline model to guide memory-bound optimizations and a DSE engine to automatically optimize compute-bound designs. In this way, we abstract the FPGA optimization process to a software level, democratizing FPGAs to a broader set of users. We validated this approach by optimizing different HPC kernels where we obtained relevant performance speedups with a minimal design effort.

References

[1] T. Nguyen, S. Williams, M. Siracusa, C. MacLean, D. Doerfler, and N. J. Wright, "The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 8–19.

[2] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 409–420.

[3] 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," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. PP, no. 99, pp. 1–1, 2016.

[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 (TCAD)*, vol. 30, no. 4, 2011.

[5] D. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *Queue*, vol. 11, no. 2, p. 40–52, Feb. 2013.

[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.

[7] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin *et al.*, "A novel multi-level integrated roofline model approach for performance characterization," in *International Conference on High Performance Computing*. Springer, 2018.

[8] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. Malas, J.-L. Vay, and H. Vincenti, "Applying the roofline performance model to the intel xeon phi knights landing processor," in *International Conference on High Performance Computing*. Springer, 2016, pp. 339–353.

[9] C. Yang, T. Kurth, and S. Williams, "Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system," *Concurrency and Computation: Practice and Experience*, 11 2019.

[10] N. Ding and S. Williams, "An instruction roofline model for GPUs," in *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2019.

[11] B. Da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools," *International Journal of Reconfigurable Computing*, vol. 2013, 2013.

[12] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007, pp. 1–12.

[13] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[14] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice boltzmann simulation optimization on leading multicore platforms," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–14.

[15] S. Muralidharan, K. O'Brien, and C. Lalanne, "A semi-automated tool flow for roofline anaylsis of opencl kernels on accelerators," in *First International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC'15)*, 2015.

[16] J. de Fine Licht, K. F. Larsen, T. Hoefler, and S. Ramos, "Modeling and Implementing High Performance Programs on FPGA," Master's thesis, University of Copenhagen, 2016.

[17] M. P. Yali *et al.*, "FPGA-Roofline: An Insightful Model for FPGA-based Hardware Acceleration in Modern Embedded Systems," Master's thesis, Virginia Polytechnic Institute, 2015.

[18] 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," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 430–437.

[19] G. Zhong, A. Prakash, Y. Liang *et al.*, "Lin-analyzer: A High-level Performance Analysis Tool for FPGA-based Accelerators," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16. New York, NY, USA: ACM, 2016, pp. 136–142.

[20] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *SIGARCH Comput. Archit. News*, 1995.

[21] 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.

[22] J. Cong *et al.*, "A study on the impact of compiler optimizations on high-level synthesis," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012.

[23] M. Siracusa and F. Ferrandi, "Tensor optimization for high-level synthesis design flows," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[24] J. M. Codina *et al.*, "A Comparative Study of Modulo Scheduling Techniques," in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS '02. New York, NY, USA: ACM, 2002.

[25] R. Tessier and H. Giza, "Balancing logic utilization and area efficiency in FPGAs," in *International workshop on Field Programmable Logic and Applications*. Springer, 2000, pp. 535–544.

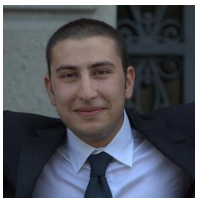
[26] E. Del Sozzo, M. Rabozzi, L. Di Tucci, D. Sciuto, and M. D. Santambrogio, "A Scalable FPGA Design for Cloud N-Body Simulation,"

in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2018, pp. 1–8.
[27] Del Sozzo, Emanuele and Rabozzi, Marco and Di Tucci, Lorenzo, "N-body open-source software implementation." [Online]. Available: https://github.com/emanueledelsozzo/NBodySimulationFPGA/blob/master/sw_version/main.c
[28] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences." in *Journal of molecular biology*, vol. 147 1, 1981.
[29] L. Di Tucci, K. O'Brien, M. Blott, and M. D. Santambrogio, "Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 716–721.
[30] L. McGuffin *et al.*, "Improving the Mapping of Smith-Waterman Sequence Database Searches onto CUDA-Enabled GPUs," *BioMed Research International*, 2015.
[31] Y. Liu *et al.*, "SWAPHI: Smith-waterman protein database search on Xeon Phi coprocessors," in *2014 IEEE International Conference on Application-Specific Systems, Architectures and Processors*, 2014.
[32] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–11.
[33] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2019.
[34] A. Parravicini, F. Sgherzi, and M. D. Santambrogio, "A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA," *arXiv preprint arXiv:2009.10443*, 2020.
[35] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

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.

Emanuele Del Sozzo got his Ph.D. in Information Technology from Politecnico di Milano in 2019. He received his B.Sc. and M.Sc. in Computer Engineering from Politecnico di Milano in 2012 and 2015 respectively. He also receives in 2015 M.Sc. degree in Computer Science from the University of Illinois at Chicago (UIC), and Alta Scuola Politecnica Diploma. His research focuses on reconfigurable architectures, code generation and optimization. He is currently a PostDoc at Politecnico di Milano.

Marco Rabozzi is Co-founder of Huxelerate and got his PhD in Information Technology at Politecnico di Milano, focusing on computer-aided design tools for FPGA-based systems, reconfigurable architectures and combinatorial optimization. In 2014 He received his M.Sc. degree in Computer Science from the University of Illinois at Chicago and, in the same year, the M.Sc. in Computer Science and Engineering from Politecnico di Milano.



Lorenzo Di Tucci is Co-founder of Huxelerate and he is pursuing his PhD in Information Technology at Politecnico di Milano, focusing on High Performance Computing and Hardware Architectures. He obtained his bachelor and master degree in engineering of computing systems from Politecnico di Milano in 2013 and 2016 respectively. In 2016 he got a M.Sc. in Computer Science from University Of Illinois at Chicago.

Samuel Williams Dr. Samuel Williams is a senior scientist in the Performance and Algorithms Research Group at the Lawrence Berkeley National Laboratory (LBNL). His research interests include high-performance computing, performance modeling, auto-tuning, computer architecture, and hardware/software co-design. Dr. Williams received his PhD in Computer Science from the University of California at Berkeley (UCB) in December of 2008 and his masters in December of 2003. During this period, his doctoral research focused on multicore architectures and automated performance tuning under the guidance of David Patterson. To that end, he created the Roofline Model to enable developers, computer scientists, computer architects, and applied mathematicians to quickly and visually assess performance bottlenecks on multicore, manycore, and GPU-accelerated systems.

Donatella Sciuto received her Laurea in Electronic Engineering from Politecnico di Milano and her PhD in Electrical and Computer Engineering from the University of Colorado, Boulder, and an MBA from Bocconi University. She is currently the Executive Vice Rector of the Politecnico di Milano and Full Professor in Computer Science and Engineering. Her main research interests cover the methodologies for the design of embedded systems and multicore systems. She has published over 300 scientific papers.

She is a Fellow of IEEE and has served as President of the IEEE Council of Electronic Design Automation from 2011 to 2013 and in different capacities in IEEE Committees and conferences.

Marco Domenico Santambrogio (SM'05) received the Laurea (M.Sc. equivalent) degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2004, the M.Sc. degree in computer science from The University of Illinois at Chicago, Chicago, IL, USA, in 2005, and the Ph.D. degree in computer engineering from the Politecnico di Milano, in 2008. He was a Post-Doctoral Fellow with the Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA. He held visiting positions with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL, USA, in 2006 and 2007, and the Heinz Nixdorf Institut, Paderborn, Germany, in 2006. He has been with the NECST Laboratory, Politecnico di Milano, where he founded the Dynamic Reconfigurability in Embedded System Design project in 2004 and the CHANGE project in 2010. He is currently an Assistant Professor with the Politecnico di Milano. His current research interests include reconfigurable computing, self-aware and autonomic systems, hardware/software co-design, embedded systems, and high performance processors and systems. Dr. Santambrogio is a Senior Member of the Association for Computing Machinery.