Exact and Heuristic Allocation of Multi-kernel Applications to Multi-FPGA Platforms

Exact and Heuristic Allocation of Multi-kernel Applications to Multi-FPGA Platforms

Junnan Shan[†], Mario R. Casu[†], Jordi Cortadella[‡], Luciano Lavagno[†], and Mihai T. Lazarescu[†]
[†]Department of Electronics and Communications, Politecnico di Torino, Torino, Italy
[‡]Department of Computer Science, Universitat Politècnica de Catalunya, Barcelona, Spain
{junnan.shan,mario.casu,luciano.lavagno,mihai.lazaescu}@polito.it,jordi.cortadella@upc.edu

ABSTRACT

FPGA-based accelerators demonstrated high energy efficiency compared to GPUs and CPUs. However, single FPGA designs may not achieve sufficient task parallelism. In this work, we optimize the mapping of high-performance multi-kernel applications, like Convolutional Neural Networks, to multi-FPGA platforms. First, we formulate the system level optimization problem, choosing within a huge design space the parallelism and number of compute units for each kernel in the pipeline. Then we solve it using a combination of Geometric Programming, producing the optimum performance solution given resource and DRAM bandwidth constraints, and a heuristic allocator of the compute units on the FPGA cluster.

1 INTRODUCTION

Field Programmable Gate Arrays (FPGAs) are an increasingly important target for parallel algorithm implementation due to their energy efficiency, flexible reconfigurability, and fast time-to-market. They promise to offer (almost) software-like programmability with (almost) GPU-like performance and (almost) ASIC-like energy efficiency. They can thus provide a sweet spot for large datacenters, where energy is a main part of overall cost. These datacenters execute a broad class of "embarassingly parallel" and widely used applications like Machine Learning (e.g. Convolutional Neural Networks, Deep Neural Networks), finite element analysis, and so on. For this reason, cloud providers like Amazon, Microsoft and Alibaba have recently offered Virtual Machines that contain several FPGAs and that can be used to accelerate datacenter-class applications with GPU-like performance at a fraction of the energy cost. These applications can now be synthesized into a bitstream by compilers like Xilinx SDAccel and SDSoc, and Intel SDK for OpenCL.

A large amount of past work addresses application implementation on a single FPGA via both RTL design and High-Level Synthesis (HLS). However, application resource requirements may often exceed those available on a single FPGA, hence multi-FPGA implementations need to be adopted, e.g., by assigning different CNN layers to different FPGAs.

In this paper we exploit an OpenCL-like (but not OpenCL-limited) execution model. In this model, an application is typically (but not always) a linear task-level pipeline of kernels, each kernel being composed of a very large number of independent Compute Units (CU). Each CU in turn contains loops which can be unrolled and pipelined to offer further parallelization. Kernels communicate among each other and with the CPU-bound "host code" via large buffers allocated in external DRAM. The designer must ensure that CUs do not interfere with each other when writing into these buffer, i.e. CU-level parallelism can be arbitrarily increased via replication. This computational model can also be supported by C++-based synthesis tools (in fact, we model our applications in C++ in order to have better control over loop handling during HLS), and fits very well many datacenter applications, like CNNs or other Neural Networks and Machine Learning algorithms.

However, globally optimizing the throughput of a task-level pipeline of kernels over multiple FPGAs is far from trivial. One must take into account simultaneously:

(1) *throughput* matching among multiple kernels, which can be increased or decreased by changing either the number of CUs or the parallelism of each CU (e.g., via unrolling);
(2) the amount of *resources and external DRAM bandwidth* used on each FPGA, which increases as more CUs are allocated to them.

The number of choices to evaluate, and hence the designer expertise and effort needed, quickly grows out of control. Note that while this problem superficially resembles the classical pipeline scheduling problem in HLS, the actual model is much more complex, because CUs that implement kernels:

(1) have many more implementation choices (e.g., via unrolling or other HLS transformations [6]) than typical Functional Units.
(2) have a multi-dimensional cost function including performance, memory bandwidth, and FPGA resources (DSPs, LUTs, FFs, and BRAMs).

In this paper, we propose a new optimization method for the implementation of task-level pipelined applications on multiple FPGAs. We assume that all communication is performed via off-chip DRAM, which is essentially the above-mentioned OpenCL inter-kernel communication model. In this scenario, our method can be used to choose how many CUs should be allocated for each kernel. This is a simple option that can be passed to FPGA compilation environments like Xilinx SDAccel, Intel SDK for OpenCL, and so on. While a mix of on-chip and off-chip communication resources would allow the exploration of an even larger design space, they are not yet supported by any of these design environments. Hence their analysis is left to future work.

Our work is fully general, and could be applied (1) to other task-level pipelined applications beyond *CNNs*, (2) to other cloud-based or super-computing FPGA platforms beyond *Amazon Web Services (AWS) F1 instances*, and (3) to other design environments beyond *SDAccel*. However, we use this generally available and well-known trio to demonstrate and quantitatively evaluate our results.

In this paper we use two Convolutional neural networks, AlexNet [5] and VGG16 [7]. Note that *our algorithms do not depend at all on the considered networks*, and these two examples are used only for the sake of illustration. Each CNN is composed of several convolutional, pooling, normalization and fully connected layers, and each convolutional layer is mapped to a kernel. As discussed in [10], we use loop tiling to reuse both the input feature maps and the weights. Memory access is optimized by reshaping the input and output feature map arrays and the weight array, to allow burst mode data transfers.

In these applications, throughput (i.e. processed images per second) is the main measure of performance, while *overall latency* (i.e. total pipeline depth) is much less important. Hence we focus on *minimizing the maximum latency among all kernels*, because it determines the Initiation Interval (II) of the pipeline, and therefore its throughput. Note also that memory bandwidth of external DRAM can be a major factor limiting the performance of memory-intensive applications like CNNs. Hence our cost and performance model takes this aspect explicitly into account.

Our flow starts from CNN models which have already been partitioned into kernels and individually optimized for FPGA implementation. Then we collect cost, memory bandwidth, and performance (throughput and latency) data from each kernel, by running several versions of its CUs, with varying degrees of parallelism, on an AWS F1. We then use these values to formulate an optimization problem that is discussed in Section 3.1 and models the multi-kernel multi-FPGA resource- and bandwidth-constrained allocation problem. This problem can then be solved:

(1) either directly by a Mixed-Integer Non-Linear Programming (MINLP) solver, to provide an exact solution in a potentially very long execution time.
(2) or indirectly by combining the power of a Geometric Programming (GP) solver, which is followed by an efficient integer relaxation of the problem variables, with a novel allocation algorithm that:
   - discretizes the result of the GP solver, and
   - tries to cluster CUs for a kernel on the same FPGA, to simplify the communication coordinated by the host code.

The second method achieves essentially the same level of optimality as the MINLP solver (whenever the latter is able to complete), in a fraction of the time.

We designed our GP model and allocator to optimize the assignment of Compute Units on multiple FPGAs while keeping into account the limitations of modern FPGAs (e.g. the maximum DRAM bandwidth), so that it can handle the large size of typical state-of-the-art CNN applications. Our contributions are:

(1) The definition of the multi-FPGA CU allocation problem for linear kernel pipelines and its constraints.
(2) The definition of a Non-Linear Programming model for that problem, and its solution both (1) by an exact (very expensive) MINLP solver and (2) by a GP solver, finding an optimal non-integer solution, followed by an allocator aimed at minimizing the spreading of CUs of one kernel to multiple FPGAs.
(3) The analysis of their result quality for two large CNN applications, implemented on large multi-FPGA AWS F1 instances.

As mentioned, we are leaving the generalization to (less common) non-linear pipelines and to (not yet available from industrial design environments) on-chip and off-chip communication mechanisms to future work.

This is the paper organization. We review past work in Sec. 2 and define the optimization problem and our heuristic in Sec. 3. Experimental results are reported in Sec. 4 and conclusions in Sec. 5.

2 RELATED WORK

Efficient allocation of processes from streaming applications to processors and accelerators is a well-studied problem in the community of compilers for parallel architectures. For example, [4] defines three levels of parallelism (task, data and pipeline) that are also exploited in our underlying execution model (tasks are called "kernels", data parallelism is exploited both at the CU level and and the loop unrolling level within a CU, and innermost loops are pipelined). Their compiler, based on the StreamIt language, is aimed at processors (the RAW machine) rather than FPGAs. Moreover, it makes only heuristic choices for allocation, while we first find an optimal non-integer solution and we relax it.

Similarly, [8] uses multiple process instances, but focuses only on process replication and FIFO allocation, while we include resources as a primary aspect of our cost function and consider array-based communication, rather than FIFO-based. Array-based is a more natural programming model, because it is supported by languages like C, C++ and OpenCl, and it requires fewer changes to legacy code, without complex logic for forking and joining data to and from data parallel CUs. More recently, [9] includes, like in our case, an explicit memory model, but solves the problem heuristically with a clustering algorithm (using ILP only as a reference), while we start from a GP relaxation for our heuristic.

On the FPGA implementation side, [3] schedules a task-parallel Static Dataflow Graph with multiple CU instances, leading to a very efficient scheduling formulation as a Set of Difference Constraints. However, it is also limited to FIFO-based communication and it does not consider multi-FPGA allocation and the resulting trade-offs.

Finally, [6] models the application as a Timed Marked Graph and uses Petri net theory to find the best overall throughput, then imposing a throughput constraint on every process and trying to satisfy it via High-Level Synthesis. However, there is no guarantee that the requested throughput is feasible, hence iterating is needed to explore the entire Pareto-optimal design space. Moreover, it does not discuss memory bandwidth nor allocation to FPGAs.

3 MULTI-FPGA OPTIMIZATION

We consider an application as a set K of kernels organized in a linear pipeline. As mentioned above, CNNs represent a relevant example, in which the kernels are the convolutional, pooling and normalization layers[1]. Each kernel workload is assigned to one or

[1]Some max-pooling layers are merged with the previous convolutional layer, whenever this allows us to optimize memory access. We do not implement the fully connected

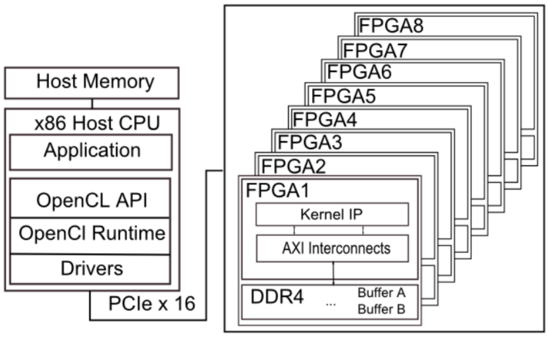

Figure 1: Architecture of AWS F1 instance.

more *compute units* (CUs) that operate concurrently. The kernels communicate through the host CPU. Since the control unit on the CPU side is quite efficient, we do not consider the CPU time in our model. Application throughput is the inverse of the pipeline initiation interval (II), which depends on the execution time of the slowest pipeline stage.

Let us define WCET_k the worst case execution time of kernel k obtained with only one CU. We consider kernels that are inherently parallel and for which the execution time ET_k scales proportionally to the number N_k of CUs for that kernel:

$$\text{ET}_k = \frac{\text{WCET}_k}{N_k}, \quad \forall k \in K \tag{1}$$

$$\text{II} = \max_{k \in K} \text{ET}_k. \tag{2}$$

To minimize II it is necessary to find the optimal value of N_k under specific constraints. We consider FPGA resource and memory bandwidth constraints, but we do not consider (yet) power constraints.

As an additional design exploration knob, we can deploy an application onto one or more FPGAs of a multi-FPGA board like the AWS F1 instance, which includes eight Xilinx UltraScale Plus FPGAs. This is also the FPGA platform where we run our experiments. In this platform, a host CPU orchestrates the execution of the kernels. Fig. 1 shows the architecture of the F1 instance. Tab. 1 summarizes variables and constants used in the problem.

The design goal is therefore not just determining the optimal N_k, but also how these CUs are allocated on F FPGAs. If we define $n_{k,f}$ as the CUs of kernel k on FPGA f, we have

$$N_k = \sum_{f=1}^{F} n_{k,f}, \forall k \in K. \tag{3}$$

Since we assume a uniformly accessed global memory, in our model a kernel execution time depends on the number of CUs but not on where they are allocated. However, keeping the CUs of a kernel in the same FPGA simplifies the host code (each pair of kernels needs only one buffer to communicate). To account for this, we introduce a *spreading* function that is minimal when all CUs of a kernel are allocated on one FPGA:

$$\phi_k = \sum_{f=1}^{F} \frac{n_{k,f}}{1+n_{k,f}} \quad \forall k \in K. \tag{4}$$

To minimize the global II and the spreading of the CUs we formulate the optimization problem shown in the following.

layers, since we are simply interested in showing a design methodology with a realistic use case, rather than benchmarking a full application.

3.1 Problem Formulation

We can combine II and spreading objectives linearly with two weights α and β into a single goal function g to minimize. The problem is then formulated as a non-linear problem with both integer and real variables:

$$\text{minimize } g = \alpha \cdot \text{II} + \beta \cdot \phi \tag{5}$$

subject to

$$\text{II} \geq \text{ET}_k, \quad \forall k \in K \tag{6}$$

$$\phi \geq \phi_k, \quad \forall k \in K \tag{7}$$

$$N_k \geq 1, \quad \forall k \in K \tag{8}$$

$$\sum_{k=1}^{|K|} n_{k,f} R_k \leq R, \quad f = 1, 2, \ldots, F \tag{9}$$

$$\sum_{k=1}^{|K|} n_{k,f} B_k \leq B, \quad f = 1, 2, \ldots, F \tag{10}$$

The constraint (8) guarantees at least one CU per kernel. In (9) and (10), R_k and B_k are resource and memory bandwidth utilization, respectively, of each CU of kernel k: in each FPGA, their sum over all kernels should not exceed R and B, the total resources and bandwidth of a single FPGA.

Table 1: Notations used in the model

Notation	Description
K	set of kernels
k	index of kernels, $1, 2, \ldots, \|K\|$
f	index of FPGAs, $1, 2, \ldots, F$
WCET_k	constant; latency of kernel k with one CU
ET_k	variable; latency of kernel k with N_k CUs
R_k	constant; FPGA resources used by one k's CU
B_k	constant; FPGA bandwidth used by one ks CU
R	constant; resource limitation in one FPGA
B	constant; bandwidth limitation in one FPGA
$n_{k,f}$	variable; CUs of kernel k allocated to FPGA f
N_k	variable; sum of $n_{k,f}$ over all the FPGAs
ϕ_k	variable; spreading function of kernel k
ϕ	variable; global spreading function
II	variable; initiation interval

3.2 Heuristic Solution

The optimization problem formulated in (5)-(10) can be solved by a Mixed-Integer Non-Linear Programming (MINLP) solver. This can lead, however, to a very long optimization time for designs with many kernels and FPGAs. Consider, for instance, that the VGG-net convolutional neural network with 20 layers spread on 8 FPGA has 160 integer variables. Especially for design space exploration, when the optimization may be repeated several times, running a MINLP solver within an exploration loop might turn out to be prohibitive.

For this reason, we propose a heuristic formulation that separates the optimization in two steps. The first step determines the total number of CUs for each kernel to minimize II. The second step allocates the CUs to the available FPGAs.

3.2.1 First Step: Geometric Programming. If we disregard the spreading minimization, i.e. $\beta = 0$ in (5), and relax the problem by letting $n_{k,f}$ take real values, the problem becomes fully symmetric across the F identical FPGAs. This implies a symmetric solution with an equal distribution of the CUs across the F FPGAs.

Let us define $\hat{n}_k \in \mathbb{R}$ the CUs that would be equally distributed. The total number of CUs of kernel k will be

$$\hat{N}_k = F \cdot \hat{n}_k. \tag{11}$$

Since we want to guarantee that at least one CU is instantiated per kernel, i.e. $\hat{N}_k \geq 1$, it is possible that $\hat{n}_k = \hat{N}_k / F$ be less than one[2].

Kernel execution time and II become

$$\hat{\mathrm{ET}}_k = \frac{\mathrm{WCET}_k}{\hat{N}_k}, \quad \forall k \in K \tag{12}$$

$$\hat{\mathrm{II}} = \max_{k \in K} \hat{\mathrm{ET}}_k. \tag{13}$$

We can thus reformulate the problem (5)-(10) with $\beta = 0$ as follows:

$$\text{minimize } \hat{g} = \hat{\mathrm{II}} \tag{14}$$

$$\text{subject to}$$

$$\hat{\mathrm{II}} \geq \hat{\mathrm{ET}}_k, \quad \forall k \in K \tag{15}$$

$$\hat{N}_k \geq 1, \quad \forall k \in K \tag{16}$$

$$\sum_{k=1}^{|K|} \frac{\hat{N}_k}{F} R_k \leq R, \tag{17}$$

$$\sum_{k=1}^{|K|} \frac{\hat{N}_k}{F} B_k \leq B. \tag{18}$$

Note that the number of unknowns $\hat{N}_k$ is F times less than the number of unknowns $n_{k,f}$ in the original formulation.

The minimization of $\hat{\mathrm{II}}$ in (14)-(18) is compatible with a Geometric Programming (GP) formulation. GP problems are solved quickly even with hundreds of variables. Therefore, we use a GP solver as the first step in our heuristic to determine $\hat{N}_k$ for all kernels.

3.2.2 Second Step: FPGA Allocation. Before allocation, the variables $\hat{N}_k \in \mathbb{R}$ must be discretized so as to obtain $N_k \in \mathbb{N}$. The integrality is enforced by a branch-and-bound technique similar to those used in ILP. Two subproblems are generated with $N_k \leq \lfloor \hat{N}_k \rfloor$ and $N_k \geq \lceil \hat{N}_k \rceil$. The search is pruned when the cost of a subproblem is greater than the best cost found. Even though this branch-and-bound technique may lead to a worst-case exponential branching tree, in practice this does not lead to excessive execution times due to the pruning strategy and the fact that the number of kernels is limited (e.g. around 20 for the VGG benchmark). The MINLP approach, on the other hand, must discretize every variable, and hence may potentially have a much larger branching tree.

For simplicity, from now we use the general term *resource constraint* to refer to both actual resource and bandwidth constraints.

The N_k CUs are allocated with a greedy heuristic. The rationale is to *allocate the critical kernels first*. These are the kernels for which a CU reduction has a significant impact on II, hence they should *all* be allocated. After each allocation of a kernel, either full or partial, the kernels are sorted in decreasing criticality order. Moreover, by sorting the FPGAs after each allocation in increasing order of resource slack, the heuristic tends to consolidate the kernels by allocating *all* the CUs to already occupied FPGAs while not exceeding the resource constraints. If it is not possible to allocate all of them, the heuristic allocate as many CUs as possible starting from the least occupied FPGA.

[2] We can liken n_k to the *average* number of CU of kernel k across F FPGAs.

The pseudo-code of the heuristic is shown in Alg. 1. We search for possible solutions in the vicinity of the initial resource constraint R used in the GP step. We define T as the maximum deviation from the initial constraint. We define Δ as the step by which the current resource constraint R_c, initialized as R, is updated at each iteration, i.e. $R_c = R_c + \Delta$. The iterations continue while $R_c < R + T$.

The for loop at line 11 partially allocates the CUs of kernels that cannot fit in one single FPGA, if any. The for loop at line 23 attempts to allocate all of the remaining CUs starting from the most occupied FPGA (while loop at line 26) and, if not possible, it allocates as many CUs as possible in the least occupied FPGA (lines 33-36).

Algorithm 1: Pseudo-code of heuristic allocation

```
1  procedure AllocateCUs(N_k, T, R, Δ)
2      CU = (CU_1, CU_2, ..., CU_|K|)            // Vector of kernel CUs to allocate
3      CU_k = N_k, ∀k                  // CUs to allocate initialized to GP values
4      R_c = R                    // FPGA resource constraint initialized to GP value
5      S = (S_1, S_2, ..., S_F)                     // Vector of FPGA resource slack
6      S_f = R, ∀f          // FPGA resource slack initialized to constraint value
7      n_{k,f} = 0, ∀k, f                        // Allocated CUs initialized to zero
8      alloc = FALSE
9      while R_c < R + T and not alloc do
10         sortCU(CU,K)                      // Sort kernels by descending criticality
11         for k = 1 to |K| do // Allocate large kernels first
12             f = 1
13             while CU_k · R_k > R do
14                 if S_f = R then
15                     δCU = ⌊R/R_k⌋
16                     CU_k = CU_k − δCU
17                     S_f = S_f − δCU · R_k
18                     n_{k,f} = n_{k,f} + δCU
19                 else
20                     f = f + 1
21         sortCU(CU,K)
22         sortFPGA(S)                            // Sort FPGAs by increasing slack
23         for k = 1 to |K| do // Allocate all kernels
24             partial_alloc = FALSE
25             f = 1
26             while f ≤ F and not partial_alloc do
27                 if S_f ≥ CU_k · R_k then
28                     S_f = S_f − CU_k · R_k
29                     n_{k,f} = n_{k,f} + CU_k
30                     CU_k = 0
31                     partial_alloc = TRUE
32                 f = f + 1
33             if CU_k > 0 then
                   // Use the space of least used FPGA (F), if possible
                   δCU = ⌊S_F/R_k⌋
34                 CU_k = CU_k − δCU
35                 S_F = S_F − δCU · R_k
36                 n_{k,F} = n_{k,F} + δCU
37             sortFPGA(S)
38         if Σ_k CU_k > 0 then
39             R_c = R_c + Δ
40         else
41             alloc = TRUE                                // All kernels allocated
```

4 EXPERIMENTAL RESULTS

We implemented our allocation heuristic in C++ and linked it to an existing efficient GP solver [2]. To validate our optimization method we used two widely used CNNs, AlexNet [5] and VGG [7]. For AlexNet, we considered both 32-bit floating point and 16-bit fixed point versions, to which we refer in the following as Alex-16 and Alex-32, respectively. For VGG, we considered only the 16-bit fixed point version. We experimented with different numbers of FPGAs, from 2 to 8, and with different resource constraints.

Tabs. 2-3 show the results of the initial characterization of the various kernels of these applications when implemented on one FPGA of the AWS F1 instance[3]. For space reasons we report only DSP and BRAM resource use, especially because these resources are much more critical than LUTs and FFs in our experiments.

Table 2: Characterization of kernels for Alex-32 (AlexNet 32-bit floating point) and Alex-16 (AlexNet 16-bit fixed point).

	Alex-32				Alex-16			
Kernels	BRAM (%)	DSP (%)	BW (%)	WCET (ms)	BRAM (%)	DSP (%)	BW (%)	WCET (ms)
CONV1	13.07	21.24	1.3	13	10.59	4.31	1.8	5.16
POOL1	2.84	0	7.03	1.78	0.05	0	3.5	1.78
NORM1	6.1	2.11	5.7	0.839	2.53	0.06	3.1	0.78
CONV2	8.73	37.59	2.4	7.19	4.39	7.63	2.1	4.11
NORM2	7.75	2.11	3.7	0.807	6.66	0.06	2.2	0.67
CONV3	5.22	28.13	5.0	7.78	2.63	5.66	2.9	6.7
CONV4	2.13	37.5	3.7	9.08	1.91	7.55	3.2	5.06
CONV5	8.73	37.5	4.2	4.84	4.39	7.55	3.1	3.29
SUM	54.57	166.18	33.1	45.32	33.15	32.82	21.9	27.55

Table 3: Characterization of VGG kernels (16-bit fixed point).

Kernels	BRAM (%)	DSP (%)	BW (%)	WCET (ms)
CONV1	3.67	2.95	2.0	28.8
CONV2	9.97	15.14	2.1	67.8
POOL2	11.62	0.03	5.2	13.3
CONV3	9.97	15.14	2.3	22.7
CONV4	9.97	15.14	2.4	32.1
POOL4	2.94	0.03	5.1	6.9
CONV5	8.32	15.07	2.0	22.8
CONV6, 7	8.32	15.05	2.3	32.9
POOL7	1.5	0.03	5.0	3.5
CONV8	2.12	15.02	2.1	24.5
CONV9, 10	2.12	15.02	2.5	37.7
POOL10	0.05	0.01	4.0	2.1
CONV11,12,13	2.12	14.99	2.6	20.3
SUM	87.37	183.67	49.7	0.4 (s)

Before reporting the details of the comparison of our heuristic with a state-of-the-art MINLP solver [1], we report on the evaluation of the effect of changing the T parameter of the heuristic while keeping the other parameter Δ set to 1%. We report the result of this analysis for Alex-16 in Fig. 2. Similar results are obtained for Alex-32 and VGG. We observe little effect of T on the value of II across a large range of resource constraints. Therefore, the following results have all been obtained with T=0%.

We ran all our optimization algorithms on a multi-core CPU (Intel Core i7-2600 @3.40 GHz, 4 Cores, 8 Threads) with 16-GB DDR3 DRAM @1333 MHz from Micron and with Linux CentOS (release 6.10), and our FPGA accelerations on AWS F1 instances with 8 FPGAs.

Out of all our experiments we selected three representative cases of the spectrum of possible multi-FPGA implementations: Alex-16 on 2 FPGAs, Alex-32 on 4 FPGAs, and VGG on 8 FPGAs. For these three cases, Tab. 4 shows the value of the two weights α and β. These values are chosen in such a way to equalize the relative importance of II and ϕ in the optimization function g in (5).

[3]While the kernel code for AlexNet has been fully optimized, and performance results are in line with the literature, the VGG kernels have not yet been fully optimized. Again, our goal is to show how CUs can be allocated, not to discuss how their internal code can be massaged for HLS.

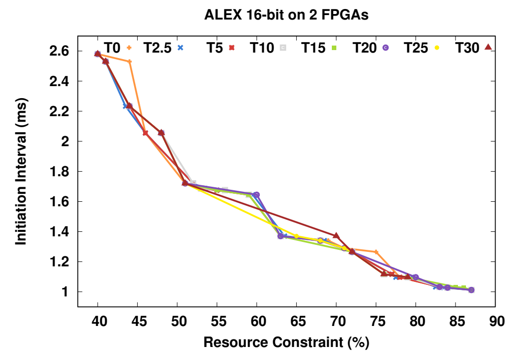

Figure 2: Alex-16 results with different values of T (in %).

Table 4: Parameters for the spreading function

Applications	α	β
Alex-16 on 2 FPGAs	1	0.7
Alex-32 on 4 FPGAs	1	6
VGG on 8 FPGAs	1	50

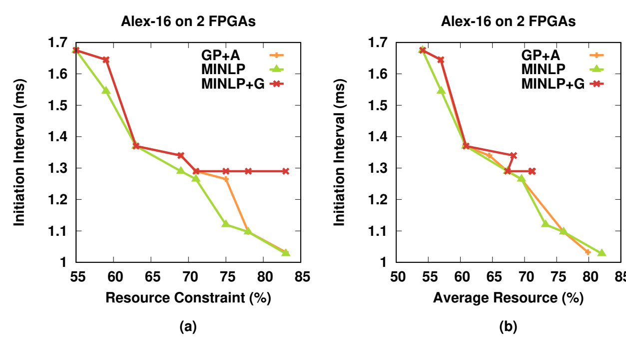

Figure 3: AlexNet 16-bit fixed-point on 2 FPGAs.

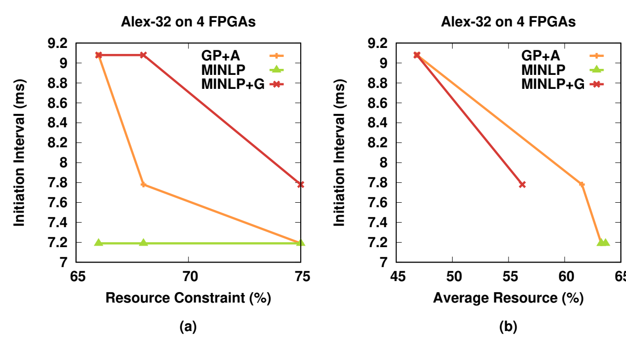

Figure 4: AlexNet 32-bit floating-point on 4 FPGAs.

The left graphs in Figs. 3-5 report the results of II obtained by changing the resource constraint, i.e. the maximum allowed FPGA resource utilization. (Incidentally, the most critical resources in all our experiments are DSPs.) The right graphs show the same points of the left graphs in a different space of II versus average FPGA resource utilization. The labels in the figure keys are as follows:

- **GP+A** refers to the heuristic consisting of GP (optimizing II) and allocation (discretizing and optimizing spreading);

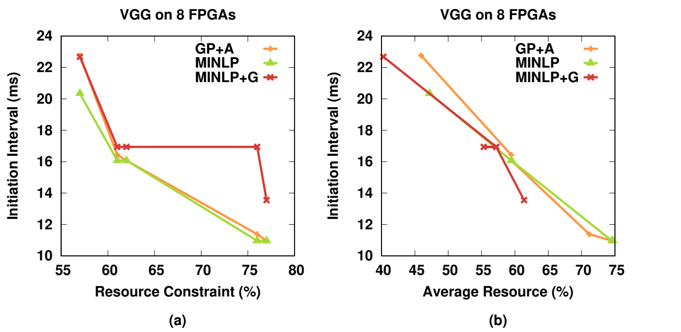

Figure 5: VGG 16-bit fixed-point on 8 FPGAs.

- **MINLP** refers to the MINLP solver set up to optimize only II and not the spreading (i.e. $\beta = 0$)[4];
- **MINLP+G** refers to the MINLP solver set up to optimize both II and spreading (i.e. α and β as in Tab. 4).

As expected, the left graphs show that MINLP obtains the best II for a given resource constraint when the spreading is ignored. With the exception of Alex-16 at low resource utilization, GP+A tracks well MINLP and in particular it catches the extremes. The results on the right graphs show that II nicely scales down as the average resource increases, especially for MINLP and GP+A.

The Alex-16 case is relevant because it shows that in some cases especially in the lower range of resource constraint, GP+A cannot reach the same performance of MINLP, but indeed behaves more similarly to MINLP+G. This is because both GP+A and MINLP+G tend to consolidate the CUs in fewer FPGAs than what MINLP does. This might result in a performance loss—25% in Fig. 4(a) at the lowest resource constraint—but in a better average FPGA utilization: Fig. 4(b) shows around 40% less average utilization of GP+A and MINLP+G compared to MINLP at the lowest resource constraint[5].

For space limitations we report only one example of resource distribution in Fig. 6, which refers to the VGG case with a specific resource constraint of 61%. The histograms show how the kernels are distributed across 8 FPGAs and how many resources each kernel uses while respecting the 61% resource constraint (SLACK ≥39% in figure). As expected from the previous discussion, both GP+A and MINLP+G tend to concentrate the kernels in one FPGA, whereas MINLP spreads them across multiple FPGAs.

Finally, the CPU time of GP+A ranges between 0.78 s (Alex-16 on 2 FPGAs) to 4.4 s (VGG on 8 FPGAs), whereas that of MINLP and MINLP+G ranges from around one minute to several hours, with a speedup that ranges from around 100x to around 1000x. The quality of the results and the low CPU time clearly show that our heuristic approach is suitable for design space exploration of multi-kernel applications deployed on multi-FPGA boards.

[4]These results show the best achievable II for a given resource constraint, but they would require *an extremely complex routing of data from each CU in one layer to several other CUs spread over multiple FPGAs, each with its own DRAM banks*, and thus they would make the host code essentially unmanageable.

[5]The three MINLP points in Fig. 4(a) represent actually the same solution, because the solver is able to reach the minimum II without saturating the resource utilization in any FPGA. This is more evident in Fig. 4(b), where the three points overlap.

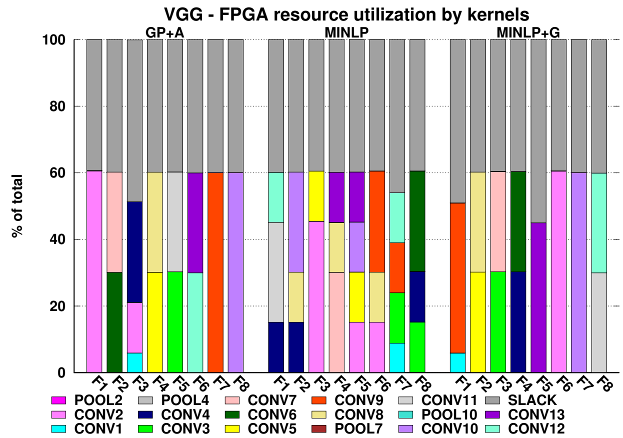

Figure 6: VGG resource usage for 61% resource constraint.

5 CONCLUSIONS

We have proposed and experimentally evaluated a new and fast method for minimizing the initiation interval of pipelined applications consisting of multiple kernels and deployed on multiple FPGAs. We optimize the number of parallel compute units (CUs) for each kernel while respecting resource and memory bandwidth constraints. The optimization problem is non-linear and with both integer (i.e. the CUs) and real variables, for which accurate MINLP solvers can be used but at the cost of unacceptable execution time. We use a two-step heuristic that first relaxes the problem by letting integer variables take real values, which allows us to use a fast geometric programming solver. Second, we discretize the results and apply a greedy allocation of the CUs over the target FPGAs, aimed at minimizing the spreading of a kernel over FPGAs. We obtain results that are comparable to what a MINLP solver can obtain, but our algorithm is 2-3 orders of magnitude faster.

REFERENCES

[1] P. Belotti. 2018. Couenne (Convex Over and Under ENvelopes for Nonline Estimation). https://www.coin-or.org/Couenne/

[2] E. Burnell and W. Hoburg. 2018. GPkit. https://github.com/convexopt/gpkit

[3] Jason Cong, Muhuan Huang, and Peng Zhang. 2014. Combining Computation and Communication Optimizations in System Synthesis for Streaming Applications. In *Proceedings of the 2014 ACM/SIGDA International Symposium on FPGAs.* 213–222.

[4] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. *SIGOPS Oper. Syst. Rev.* 40, 5 (Oct. 2006), 151–162.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.* 1097–1105.

[6] Luca Piccolboni, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2017. COSMOS: Coordination of High-Level Synthesis and Memory Optimization for Hardware Accelerators. *ACM Trans. Embed. Comput. Syst.* 16, 5s (Sept. 2017).

[7] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[8] Sander Stuijk, Marc Geilen, and Twan Basten. 2008. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *Computers, IEEE Transactions on* 57 (11 2008), 1331–1345.

[9] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. 2009. Synergistic Execution of Stream Programs on Multicores with Accelerators. In *Proceedings of ACM SIGPLAN/SIGBED 2009.* 99–108.

[10] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '15).* ACM, New York, NY, USA, 161–170.