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

Original

Availability:
This version is available at: 11583/2740832 since: 2019-07-09T14:29:32Z

Publisher:
IEEE

Published
DOI:10.1145/3316781.3317821

Terms of use:
openAccess
This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)
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 “embarrassingly 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 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. 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. 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 or 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

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:

\[
ET_k = \frac{WCET_k}{N_k}, \quad \forall k \in K
\]

(1)

\[
II = \max_{k \in K} ET_k.
\]

(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

\[
N_k = \sum_{f=1}^{F} n_{k,f}, \quad \forall k \in K.
\]

(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:

\[
\phi_k = \sum_{f=1}^{F} \frac{n_{k,f}}{R_k}, \quad \forall k \in K.
\]

(4)

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

### 3.1 Problem Formulation

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

\[
\begin{align*}
\text{minimize} \quad g &= \alpha \cdot II + \beta \cdot \phi \\
\text{subject to} \quad &II \geq ET_k, \quad \forall k \in K \\
&\phi \geq \phi_k, \quad \forall k \in K \\
&N_k \geq 1, \quad \forall k \in K \\
&\sum_{k=1}^{K} n_{k,f} R_k \leq B, \quad f = 1, 2, \ldots, F \\
&\sum_{k=1}^{K} n_{k,f} B_k \leq B, \quad f = 1, 2, \ldots, F
\end{align*}
\]

(5)-(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

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K)</td>
<td>set of kernels</td>
</tr>
<tr>
<td>(k)</td>
<td>index of kernels, (1, 2, \ldots,</td>
</tr>
<tr>
<td>(f)</td>
<td>index of FPGAs, (1, 2, \ldots,</td>
</tr>
<tr>
<td>WCET(_k)</td>
<td>constant; latency of kernel (k) with one CU</td>
</tr>
<tr>
<td>ET(_k)</td>
<td>variable; latency of kernel (k) with (N_k) CUs</td>
</tr>
<tr>
<td>(R_k)</td>
<td>constant; FPGA resources used by one (k)'s CU</td>
</tr>
<tr>
<td>(B_k)</td>
<td>constant; FPGA bandwidth used by one (k)'s FPGA</td>
</tr>
<tr>
<td>(R)</td>
<td>constant; resource limitation in one FPGA</td>
</tr>
<tr>
<td>(B)</td>
<td>constant; bandwidth limitation in one FPGA</td>
</tr>
<tr>
<td>(n_{k,f})</td>
<td>variable; CUs of kernel (k) allocated to FPGA (f)</td>
</tr>
<tr>
<td>(N_k)</td>
<td>variable; sum of (n_{k,f}) over all the FPGAs</td>
</tr>
<tr>
<td>(\phi_k)</td>
<td>variable; spreading function of kernel (k)</td>
</tr>
<tr>
<td>(\phi)</td>
<td>variable; global spreading function</td>
</tr>
<tr>
<td>II</td>
<td>variable; initiation interval</td>
</tr>
</tbody>
</table>

### 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 layers, since we are simply interested in showing a design methodology with a realistic use case, rather than benchmarking a full application.
We can thus reformulate the problem (5)-(10) with from the least occupied FPGA.

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 \( \hat{n}_k \) be less than one\(^2\).

Kernel execution time and II become

\[
\hat{E}_T_k = \frac{WCET_k}{N_k}, \quad \forall k \in K \tag{12}
\]

\[
\hat{I}_I = \max_{k \in K} \hat{E}_T_k. \tag{13}
\]

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

\[
\min \ g = \hat{I}_I \tag{14}
\]

subject to

\[
\hat{I}_I \geq \hat{E}_T_k, \quad \forall k \in K \tag{15}
\]

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

\[
\sum_{k=1}^{|K|} \hat{N}_k R_k \leq R, \tag{17}
\]

\[
\sum_{k=1}^{|K|} \hat{N}_k B_k \leq B. \tag{18}
\]

Note that the number of unknowns \( \hat{N}_k \) is \( F \) times less than the number of unknowns \( n_k \), in the original formulation.

The minimization of \( \hat{I}_I \) 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 \( \Delta \) 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

```plaintext
1 procedure AllocateCUs(\( N_k, T, R, \hat{h} \))
2 \( \hat{I}_I \) = \( \hat{I}_I \) \( \hat{E}_T_k \) \( \forall k \in K \) \( \hat{N}_k \) \( \forall k \in K \)
3 \( \sum_{k=1}^{|K|} \hat{N}_k R_k \leq R \)
4 \( \sum_{k=1}^{|K|} \hat{N}_k B_k \leq B. \)
5 while \( R_c < R + T \) and not \( \text{all done} \)
6 sortCUs(\( K \)) \( \text{// Sort kernels by descending criticality} \)
7 for \( k = 1 \) to \( |K| \) do \( \text{// Allocate large kernels first} \)
8 \( f = 1 \)
9 while \( f \) \( \leq F \) and not \( \text{partial alloc} \)
10 if \( S_f \geq \hat{C}_U \) \( \text{// FPGA resource slack initialized to zero} \)
11 \( \hat{C}_U = \hat{C}_U - \hat{C}_U \)
12 \( \hat{S}_f = \hat{S}_f - \hat{C}_U \)
13 \( \hat{n}_k = \hat{n}_k + \hat{C}_U \)
14 \( \hat{C}_U = 0 \)
15 \( \hat{I}_I = \hat{I}_I \)
16 \( f = f + 1 \)
17 \( \text{if} \) \( \hat{C}_U > 0 \) \( \text{// Use the space of least used FPGA (F), if possible} \)
18 \( \hat{C}_U = \hat{C}_U - \hat{C}_U \)
19 \( \hat{S}_f = \hat{S}_f - \hat{C}_U \)
20 \( \hat{n}_k = \hat{n}_k + \hat{C}_U \)
21 sortFPGA(\( K \)) \( \text{// All kernels allocated} \)
```

### 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).**

<table>
<thead>
<tr>
<th>Kernels</th>
<th>Alex-32 (16-bit fixed point)</th>
<th>Alex-16 (16-bit fixed point)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRAM (%)</td>
<td>DSP (%)</td>
<td>BW (%)</td>
</tr>
<tr>
<td>CONV1</td>
<td>13.07</td>
<td>21.24</td>
</tr>
<tr>
<td>POOL1</td>
<td>2.84</td>
<td>0</td>
</tr>
<tr>
<td>NORM1</td>
<td>6.1</td>
<td>2.11</td>
</tr>
<tr>
<td>CONV2</td>
<td>5.73</td>
<td>37.19</td>
</tr>
<tr>
<td>NORM2</td>
<td>5.73</td>
<td>2.11</td>
</tr>
<tr>
<td>CONV3</td>
<td>2.22</td>
<td>28.18</td>
</tr>
<tr>
<td>CONV4</td>
<td>213</td>
<td>37.2</td>
</tr>
<tr>
<td>CONV5</td>
<td>5.73</td>
<td>37.19</td>
</tr>
<tr>
<td>SUM</td>
<td>34.57</td>
<td>346.18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Kernels</th>
<th>BRAM (%)</th>
<th>DSP (%)</th>
<th>BW (%)</th>
<th>WCET (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONV1</td>
<td>5.67</td>
<td>2.95</td>
<td>2.0</td>
<td>28.8</td>
</tr>
<tr>
<td>CONV2</td>
<td>11.62</td>
<td>0.03</td>
<td>9.5</td>
<td>11.3</td>
</tr>
<tr>
<td>POOL1</td>
<td>7.97</td>
<td>15.14</td>
<td>2.3</td>
<td>22.7</td>
</tr>
<tr>
<td>CONV3</td>
<td>11.62</td>
<td>0.03</td>
<td>9.5</td>
<td>11.3</td>
</tr>
<tr>
<td>POOL2</td>
<td>2.94</td>
<td>0.03</td>
<td>9.1</td>
<td>6.9</td>
</tr>
<tr>
<td>CONV4</td>
<td>5.32</td>
<td>15.07</td>
<td>2.0</td>
<td>22.8</td>
</tr>
<tr>
<td>CONV5</td>
<td>5.32</td>
<td>15.07</td>
<td>2.0</td>
<td>22.8</td>
</tr>
<tr>
<td>POOL3</td>
<td>1.5</td>
<td>0.03</td>
<td>9.0</td>
<td>3.5</td>
</tr>
<tr>
<td>CONV6</td>
<td>2.12</td>
<td>15.02</td>
<td>2.1</td>
<td>24.8</td>
</tr>
<tr>
<td>CONV7</td>
<td>2.12</td>
<td>15.02</td>
<td>2.1</td>
<td>24.8</td>
</tr>
<tr>
<td>POOL4</td>
<td>9.05</td>
<td>0.03</td>
<td>4.0</td>
<td>2.1</td>
</tr>
<tr>
<td>CONV8</td>
<td>2.12</td>
<td>14.99</td>
<td>2.6</td>
<td>20.3</td>
</tr>
<tr>
<td>SUM</td>
<td>87.37</td>
<td>185.67</td>
<td>49.7</td>
<td>0.4 (6)</td>
</tr>
</tbody>
</table>

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 \(\Delta\) 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 \(\alpha\) and \(\beta\). These values are chosen in such a way to equalize the relative importance of II and \(\phi\) 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 CU's can be allocated, not to discuss how their internal code can be massaged for HLS.

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);
With the exception of Alex-16 at low resource utilization, GP+A would make the host code essentially unmanageable. It would require 4 solver to reach the minimum II without saturating the resource utilization in other CUs spread over multiple FPGAs, each with its own DRAM banks applications deployed on multi-FPGA boards.

As expected, the left graphs show that MINLP obtains the best II for a given resource constraint, while 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. 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.

MINLP refers to the MINLP solver set up to optimize only II and not the spreading (i.e. β = 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, while 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. 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.

REFERENCES