Kernel Merging for Throughput-Oriented Accelerator Generation

Nicolas Derumigny
Colorado State University
Fort Collins, Colorado, USA

Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP, LIG
38000 Grenoble, France

Louis-Noël Pouchet
Colorado State University
Fort Collins, Colorado, USA

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS,
Grenoble INP, LIG
38000 Grenoble, France

Abstract

Progresses in High-Level Synthesis have enabled programmers to quickly explore design spaces for high-performance accelerators (e.g. to explore trade-offs between coarse-grain and fine-grain hardware parallelism). However, resource sharing opportunities are often under-exploited by HLS tools, especially in coarse-grain pipelined designs.

In this work, we target the issue of generating multi-purpose yet efficient pipelined accelerators, demonstrating our approach on sequences of dense linear algebra computations. We develop polyhedral program analysis to generate the accelerator structure, as well as their profitability criteria. In particular, we leverage cross-loop compute unit reuse to create coarse-grain pipelined designs suited for batched execution of sequences of operations.

1 Introduction

The accessibility of accelerator design has significantly increased, following the constant improvement in quality and ease of use of the hardware/software design stack (e.g., with compilers for High-Level Synthesis such as the Xilinx Merlin compiler [6, 28], with the HLS tools themselves [14, 31], etc.). Designers can now quickly generate customized designs for a particular application, or possibly (a set of) kernels within it which are candidate for profitable acceleration [7, 33]. However, updating the functionalities being accelerated can be difficult: at best, it requires uploading a new bitstream on an FPGA, and at worst it is not possible afterwards for ASIC-based designs.

Flexible accelerators, e.g., using overlays [19] or VTA [22], is a response to the reprogrammability issue of specialized accelerators, attempting to bring the best of both worlds: (most of) the performance benefits of hardware specialization, while maintaining some generality of computations that can be accelerated. In this work, we make a simple yet pragmatic observation: *it is possible to easily build a semi-generic accelerator by restricting the functionalities addressed to those amenable to polyhedral modeling*, that is, each functionality supported (e.g., GEMM, AXPY, etc.) by the accelerator can be exactly modeled as a polyhedral program, where the loop bounds and array access functions are affine expressions made of the surrounding loop iterators.

Specifically, we demonstrate how *kernel merging* can be efficiently implemented to create a multi-functionality accelerator with high throughput and low area, when these kernels are polyhedral programs, leveraging polyhedral code generation algorithms such as CLooG [2]. We show that performance/area profitability criteria for kernels to be candidate for merging in a common accelerator can be expressed as properties of the kernels' polyhedral representation, specifically focusing on a rich mix of dense linear algebra kernels. In particular, we demonstrate how to create a generic accelerator accepting arbitrary linear algebra expressions as input (on scalars, vectors, matrices), by merging elementary polyhedral kernels for each functionality, and enabling batch-processing of expressions automatically. When computing a correlation matrix for example, this generic accelerator can achieve nearly the same throughput as a specialized fixed-function accelerator, and provides gains in terms of performance per operator usage on batched workloads by enabling more advanced resource sharing via coarse-grain pipelining compared to specialized accelerators. We make the following contributions:

- We present a system to build a throughput-oriented, multi-functionality accelerator from a set of input polyhedral kernels, each describing some elementary functionality to be supported.
- We develop polyhedral-based analysis and transformations to merge polyhedral kernels, and easily expose hardware modules that are candidate for replication and sharing.
- We conduct extensive experimental evaluation, on numerous dense linear algebra workloads, of two generic accelerators that are fully implemented and measured in-situ on a Xilinx ZCU104 board, demonstrating their competitiveness in throughput per area compared to fixed-function accelerators optimized for throughput or resource sharing, as well as accelerators generated by ScaleHLS [33].

The paper is organized as follows. Sec. 2 motivates the problem and outline our proposed solution for semi-generic accelerator design. Sec. 3 develop polyhedral analyses for the accelerator design, itself summarized in Sec. 4. Experimental

IMPACT'23, January 16, 2023, Toulouse, France
.

results are presented in Sec. 5, before discussing related work, limitations, and concluding.

2 Background and Motivation

We illustrate the gains of a semi-generic accelerator on a simple example: a workload composed of 3 successive correlation matrix (CORR) computations, a widely used data science calculus. First, we show how coarse grain pipelining may help speed up batched computation of CENTER, a subproblem of CORR, them we show the design choices at stake when crafting a semi-generic accelerator capable of efficient execution of both problems.

2.1 Example: Data Centering

```
L1: for (j = 0; j < N ; j++)
      mean[j] = 0.0;
L2: for (i = 0; i < N ; i++)
      for (j = 0; j < N ; j++)
        mean[j] += data[i][j];
L3: for (j = 0; j < N ; j++)
      mean[j] /= N;
L4: for (i = 0; i < N ; i++)
      for (j = 0; j < N; j++)
        data[i][j] -= mean[j];
```

Figure 1. CENTER naive implementation

Let us consider the program realising the following matrix transformation, corresponding to data centering:

$$X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$$

One naive implementation of this computation is given in Fig. 1 uses four loop nests with different operators:

L1: Initialisation of the mean vector (no operator)
L2: Column-wise accumulation of the matrix coefficients (+)
L3: Division of the previous accumulation by N (/)
L4: Column-wise subtraction of the mean to the input matrix (−)

These loops forms what we call *functionalities* or *kernel*, which are defined as affine subparts of the input program represented using single loop nest. Under the resource sharing point of view, some of this functionalities can rely on the same physical compute unit, that may or may not be shared across kernels. For example, operator sharing can happen between the addition and subtraction part, as it boils down to a preprocessing of a single bitflip on FPGA per FP16-encoded data. In the rest of the paper, we note this operator ±.

The dispatch of kernels over functional units that executes them is fundamental for the generation of efficient accelerators. For example, the usual coarse-grain replication [13, 17] of a single high-performance design will fail to provide the best throughput-per-area on a sequence of CENTER. However, an accelerator using 2 ± units can benefit from the low usage of / to share the compute unit inside independent problems of a batch; an example of its implementation is given in Fig. 1.

This sharing is achieve through retiming [27] of the kernels: by spreading problems across time, we avoid simultaneous usage of the / operator, enabling further resource sharing. The transformed code is reported on Fig. 2. In the HLS framework [30], this retiming must be followed by a loop merging to ensure operator reuse; a sequence of transformations that is equivalent to the creation of a coarse-grain pipeline [35], each stage of the pipeline executing one kernel.

```
for(id=0; id<BATCH_SIZE+4; id++)
  for (i = 0; i < N ; i++)
    for (j = 0; j < N ; j++) {
      if (id < BATCH_SIZE and i==0)
        mean[id][j] = 0.0;
      if (id < BATCH_SIZE+1 and id>=1)
        mean[id-1][j] += data[id-1][i][j];
      if (id < BATCH_SIZE+2 and id>=2 and i==0)
        mean[id-2][j] /= N;
      if (id >= 3 and i==0)
        data[id-3][i][j] -= mean[id-3][j];
    }
```

Figure 2. CENTER Coarse-grain pipelined implementation

However, this merging is not trivial when it comes to iteration spaces: L1 and L3 iterate over a space of size N, while L2 and L3 iterates over a space of size N^2, hence the need of conditions on the loop iterator (here i) to ensure the correct number of execution of the loops bodies of smaller iteration space. As a downside, this means that the divider unit is idle at least $(N-1)/N$ fraction of the time during the whole computation.

More generally, we can quantify the quality of the design by the mean occupancy of its units, that is, the mean occupancy over all units, weighted by their replication factor in the final design. For a fixed input graph of computation, the more the units are used, the lesser the average execution time will be. Occupancy as well as area units (expressed as DSP usage) is reported in Tbl. 1 for a coarse-grained pipelined design realising 10-batched execution of CENTER, compared to a dedicated design either replicated 10 times (CENTERx10) or 10 successive calls to the same IP (10xCENTER); CGP-CENTER-inf denoting the maximum achievable throughput, corresponding to an infinite number of successive independent CENTER instances.

While, as expected, the occupation of the divisor unit progressed by 44.6% (from 0,76% to 1.1 %), occupancy of the ± unit dropped, which leads to a lower total occupancy. This effect is due to the initial filling and emptying of the pipeline: due to the merging, each unit is idle for $3 \cdot N^2$ cycles waiting for the other stage to complete, as demonstrated by the if guards over `id` in the final code.

Benchmark	Cycles/Pb	Operators	DSP	Occupancy	Avg. Occ.
CENTER	8343	1±, 1/	2	±: 98.2% /: 0.76%	49.5%
CENTERx10	834	10±, 10/	20	±: 98.2% /: 0.76%	49.5%
10xCENTER	8343	1±, 1/	2	±: 98.2% /: 0.76%	49.5%
CGP-CENTERx10	5744	2±, 1/	4	±: 71.3% /: 1,1 %	47.9%
CGP-CENTER-inf	4096	2±, 1/	4	±: 100% /: 1.56 %	50.8%

Table 1. Performance and area metric for coarse-grained pipeline (CGP) vs coarse grained replication (CGR) of CENTER accelerator (matrices of size 64×64, FP16 data type)

This example shows that even though theoretical gains may be achieved by coarse-grain pipelining, the real-life speedup is far from being always beneficial, as a large batch ing factor is needed to compensate the initial and final sub-optimal execution stage.

2.2 A More Complex Example: Correlation

Even though CENTER transformation is a part of the Correlation computation, optimising a Correlation accelerator to the sole computation of batched sequences of CENTER is flawed as it does not take into account all required operators. Indeed, Correlation can be decomposed into several computations, corresponding to the loop nests a programmer would write when designing an HLS accelerator:

- CENTER: $X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$
- STDDEV: $\sigma^X_j = \sqrt{\sum_i (X^C_i)^2/n}$
- CENTER-REDUCE: $X^{CR}_{ij} = \left(X_{ij} - \sum_{i'} X_{i'j}\right) / (\sigma^X_j \cdot \sqrt{n})$
- T-MATMULT: $(X^{CR})^t \cdot X^{CR}$

Assuming N is the size of the input matrix, only T-MATMULT is computed in $O(N^3)$ operations, the others being computed in $O(N^2)$. Furthermore, T-MATMULT uses only additions and multiplications, which means that the majority of the time will be spent using these units on a dedicated accelerator: sharing them will only lead to marginal gains. However, CENTER, STDDEV and CENTER-REDUCE also require the use of a division and a square root operator, which can be shared between independent batched executions of Correlation. This lead to significant area gains over the traditional coarse-grain replication strategy by avoiding unnecessary replicas of low-usage units, i.e. division and square root operators, with minimal impact on overall latency.

This time, the kernel merging approach allows us to mix both sharing and replication: we replicate the MATMULT accelerator (composed of $+$ and $*$) to keep minimal impact of the sharing on the overall execution time, but we share the $/$ and $\sqrt{\cdot}$ one. This lead to an increase of 17 % of the global occupancy compared to a basic accelerator generated with no intra-batch sharing, and an increase in execution time of 10.1 % while consuming two $/$ and $\sqrt{\cdot}$ compute units less.

Furthermore, we enrich the accelerator with additional data routing capability to become *generic*: depending on a user command, any sub-computations of Correlation can be computed. Area and execution time of the GA-CORR3 (generic accelerator capable of executing a 3-batched Correlation) compared to a simple non batched, non-generic accelerator is reported in Tbl. 2. As expected, the generic batched accelerator is able to provide a reduction of 66.7 % of the number of dividers and square root, no change in terms of adders / multipliers, to the cost of a 10.1 % increase in execution time. This is best translated by the global occupation metric, which jumped up by 16.96 %.

Benchmark	Cycles	Nb of $+$ and $*$	Nb of $\sqrt{\cdot}$ and $/$	Global Occupancy
3xCORR	291221	3	3	46.78%
GA-CORR3	320603	3	1	63.74%

Table 2. Performance and area metric for coarse-grain pipelined correlation and dedicated accelerator

3 Kernel Merging For Multi-Functionalities

We now present our approach to building a multi-functionality accelerator. Intuitively, we start from a set of *polyhedral kernels*, each computing a particular functionality. We aim to capture what kind of workloads can be executed with these functionalities available, and produce a throughput-optimized accelerator implementation for those. In this work we focus on compositions of dense linear algebra kernels, although the techniques presented are not limited to this particular class of computations.

3.1 Polyhedral kernel representation

In this work a kernel is a polyhedral program, that is a program with a static control-flow (every branch taken in the code can be exactly predicted at compile-time, independently of the value of the data computed on). In addition, polyhedral programs must be described exactly using only affine functions of the surrounding loop iterator and parametric constants. Three structures are used to describe such programs: for each statement S, we define their *iteration domain* $\mathcal{D}_S$, which describe the set of all dynamic execution of the statement, each identified by the vector of values that the surrounding loop iterators take when it executes (that is the iteration vector $\vec{x}_S$); their *access functions* which maps every iteration vector to the specific memory location(s) accessed by that instance; and a *scheduling function* Θ_S which maps every iteration vector with a multidimensional timestamp, such that in the transformed code, the iteration vectors are executed in the lexicographic order of their timestamps [2, 9].

We illustrate with the two kernels below, where for the sake of illustration we decomposed a classical GEMM kernel into two kernels.

The iteration domain of Kernel1 is $\mathcal{D}_{K1} : \{[i,j] : 0 \leq i < N \; and \; 0 \leq j < N\}$, and Kernel2 is $\mathcal{D}_{K2} : \{[i,j,k] : 0 \leq i < N \; and \; 0 \leq j < N \; and \; 0 \leq k < N\}$. The access functions of K1 include $Read_{K1} : \{[i,j] \rightarrow C[x,y] : x = i \; and \; y = j\}$ and K2 includes $Read_{K2} : \{[i,j,k] \rightarrow A[x,y] : x = i \; and \; y = k\}$. The original schedule of K1 is $\Theta_{K1}(\vec{x}_{S1}) = \{[i,j] \rightarrow$

```
// Kernel 1
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
      C[i][j] = beta * C[i][j]; // S1
// Kernel 2
for (i = 0; i < N; ++i)
 for (j = 0; j < N; ++j)
   for (k = 0; k < N; ++k)
      C[i][j] += alpha * A[i][k] * B[k][j];
```

Figure 3. Example

$[t1, t2, t3, t4, t5] : t1 = 0 \text{ and } t2 = i \text{ and } t3 = 0 \text{ and } t4 = j \text{ and } t5 = 0$, that is a $2d + 1$ encoding of the schedule, for a loop depth d [10, 11].

3.2 Kernel Set and Workloads

Given a set of polyhedral kernels that are candidate to be merged, we aim to execute workloads that are arbitrary compositions (in sequence or in parallel) of calls to these kernels. These computations can be captured by a simple language for straight-line programs, which is then trivially amenable to compilation, to extract a a forest of directed acyclic graphs, where each node represents one kernel call. We assume each kernel represents a pure function, and summarizes its functionality as follows.

Definition 1 (Kernel representation)**.** *Given a kernel K, we define its functionality as the signature of the kernel augmented with the loop bounds, for each loop:*

$$K : input_1, ..., input_n, N_1, ..., N_m \longrightarrow output_1, ..., output_p$$

We also define Ops_K the set of arithmetic operations executed by K.

For example, the complete signature of $K2$ is

$$K2 : C[N][N], A[N][N], B[N][N], alpha, N, N, N \longrightarrow C[N][N]$$

where $Ops_{K2} = \{+, *, *\}$. A workload in the present work can be modeled as a straight-line program, such that (a) temporary variables are allowed; (b) there is a single kernel call per instruction; (c) type and size analysis for every input/output passed as argument to the program kernels succeeds, given the signatures of every kernel. Focusing on (dense) linear algebra, high-level expressions can be written in this simple form, which is then compiled to obtain a sequence of kernel calls implementing this program. Parallelism between kernel calls is automatically detected from the DAGs, creating "batches" of calls when possible from the input workload, simply recognizing parallelizable operations by computing the earliest schedule of each node in the DAGs.

We illustrate with the simple following program with 4 instructions, that is a valid input to our system. For clarity `K1` is renamed to `MatScale`, and `K2` is renamed to `MatMulScaleA`. In our prototype implementation, supported variable types are scalars, 1D arrays (vectors) and 2D arrays (matrices), which should all be of the same data type.

```
TMP1[N][N] := MatScale(C1[N][N],42,N,N)
TMP2[N][N] :=
MatMulScaleA(TMP1[N][N],A[N][N],B[N][N],51,N,N,N)
TMP3[N][N] := MatScale(C2[N][N],43,N,N)
TMP4[N][N] :=
MatMulScaleA(TMP3[N][N],A[N][N],B[N][N],52,N,N,N)
```

This program may be input by the user, and is then compiled to a sequence of "instructions" to be executed by the accelerator. As described in Sec. 4, the accelerator executes a stream of instructions given as input, where each instruction contains the name of the kernel to invoke, to which hardware unit it is placed, and the operands/loop bound information as in the example above. The order of execution follows the order of instructions sent to the accelerator. A simple compilation step creates this sequence of instructions from the input program above.

A simple dataflow analysis produces these two DAGs: $MatMulScaleA(MatScale(C1, 42), A, B, 51)$ and the similar $MatMulScaleA(MatScale(C2, 43), A, B, 52)$ from this input program. This analysis delivers the set of calls to be executed as their earliest start time (assuming each call takes 1 time quantum), e.g. `MatScale:0,0` and `MatMulScaleA:1,1` giving explicitly the number of calls (i.e., the number of entries per kernel name) and the parallelism opportunities (i.e., all calls at the same time step can be executed in parallel). In our current implementation, we weight timesteps by their iteration latency, and a simple greedy placement of the calls on the available hardware units is implemented.

Therefore the problems to be addressed when designing the accelerator include (a) how many *parallel instances* of each kernel should be possible? And (b) Which operations (+,*, etc) may be shared between kernels?

3.3 Kernel Merging

We now outline our high-level method for generating a semantically correct perfectly nested loop structure, that capture the set of all functionalities to be implemented by the accelerator. We leverage polyhedral program analysis and transformations [2] to create such code structure.

Iteration domain extension The first operation is to normalize all kernels so that every statement is represented by an iteration domain of identical, maximal dimensionality across all kernels, while preserving the semantics. This amounts to computing a maximal common loop embedding, and statement perfectization [33] is an instance of such transformation. Specifically, we first compute $maxd$ the maximal dimensionality of all iterations domains to be merged: $maxd = max_{K \in kernels} dim(\mathcal{D}_K)$. Then, for every kernel whose dimensionality is less than $maxd$, we create $\mathcal{D}_K^{ext} = Universe(maxd) \cap \mathcal{D}_K \cap oneiterdims(K)$, where $Universe(x)$ builds the infinite/unbounded polyhedron of dimensionality x, and $oneiterdims(K)$ is the lexicographic minimum of every dimension in *maxd* that is not a dimension in $\mathcal{D}_K$. For example, we would get: $\mathcal{D}_{K1}^{ext} : \{[i,j,k] : 0 \le i < N \textit{ and } 0 \le j < N \textit{ and } k = 0\}$.

We then further extend the iteration domains systematically with one additional dimension: *kid*, which represents the unique ID of a kernel that is merged. For our example, assuming Kernel1 (K1) identifier is 1, and K2's is 2, we get: $\mathcal{D}_{K1}^{ext} : [K1] \rightarrow \{[kid,i,j,k] : 0 \le i < N \textit{ and } 0 \le j < N \textit{ and } k = 0 \textit{ and } kid = K1\}$.

Scheduling for fusion and pipelining The next operation builds the union of all extended iteration domains into a single polyhedral program, by building a schedule for fusion. This schedule merges all loop levels, and only separate kernels at the inner-most loop level. For example, the short notation for Θ_{K2} is $\{[i,j,k] \rightarrow [0,i,0,j,0,k,0]\}$. The schedules merging K1, then K2, are simply their original identity schedule (possibly extended to *maxd*), where we use the kernel id to compute the last schedule dimension, for every statement in each kernel. We have $\Theta_K : [K] \rightarrow \{[kid,i,...,m] \rightarrow [0,i,0,...,0,m,kid]\}$ if the kernel contains a single statement, otherwise *kid* needs to be extended to model the unique id of every statement in the kernel instead, in their order of execution, such that for every kernel and every statement *kid* is globally unique.

For example, to fuse K1 with K2 we would get $\Theta_{K1}^{ext} : [K1] \rightarrow \{[kid,i,j,k] \rightarrow [0,i,0,j,0,k,kid] : kid = K1\}$, and $\Theta_{K2} : [K2] \rightarrow \{[kid,i,j,k] \rightarrow [0,i,0,j,0,k,kid] : kid = K2\}$. Note however further modification of the schedule may be implemented: in particular, *loop permutation* may be employed to implement fine-grain parallelism when possible, as discussed below in Sec. 3.4, for example $\Theta_{K2} : [K2] \rightarrow \{[i,j,k] \rightarrow [0,i,0,k,0,j,kid] : kid = K2\}$ permutes the k and j loops, to expose a synchronization-free inner-parallel loop if possible.

Controlling separation The final operation is to actually generate the candidate loop nest, by using polyhedral code generation [2]. Intuitively, CLooG [2] generates a code that scans the iteration domains in the lexicographic order of the timestamps computed by the Θ functions. A key aspect of performance for the generated codes is to implement *separation* along every loop dimension, that is the process of grouping iterations of the loop as a function of the specific set of statements to be executed. For example, along the k loop, at iteration 0 both K1 and K2 execute, but at iteration > 0 only $K2$ executes. In this work, we aim to push conditionals that guard the execution of a statement to the inner-most loop level, therefore we simply turn off separation in CLooG, to obtain the code below:

```
for (i = 0; i < N; ++i)
 for (k = 0; k < N; ++k)
   for (j = 0; j < N; ++j) {
      if (KER == K1 && k == 0)
         C[i][j] = beta * C[i][j]; // S1
      if (KER == K2)
         C[i][j] += alpha * A[i][k] * B[k][j];
   }
```

Figure 4. Example code structure

3.4 Profitability Criteria

While any set of polyhedral programs can be merged with the procedure above, not all such programs are candidate for *efficient* acceleration, and may not benefit from being merged with other kernels. However, the profitability criteria can be expressed as the result of polyhedral analyses on the set of kernels.

Pipelining A central objective is to enable coarse-grain pipelining across kernels. Therefore we model a criterion for making pipelining *possible* (otherwise no pipelining is implemented), that eventually drives the loop order: the inner-most loop should be such that either there is no loop-carried dependence (LCD) along it for the kernel, or if there is a LCD, the distance must be constant, and greater than the expected iteration latency (for one iteration of the inner-most loop). The final loop permutation for the program is computed such that we minimize dependences satisfied by the inner-most loop level in the merged program, using only loop permutations as the possible transformations. We simply compute all possible loop permutations for the merged loop nest, and for each case compute whether the inner loop is parallel. If this system has no solution, we relax it to enable LCD for the inner-most loop level iff the dependence distance is greater than the iteration latency for the statement.

Exposing Functional Units A kernel can be viewed as the actual computation statement(s) associated with it, along with their iteration domain. As we generate a fused loop nest, all statements share the same unique loop nest implemented in hardware to iterate them. Therefore two parallel instances of a kernel can be implemented by simply replicating the *statement(s)* in the inner-most loop. We call such hardware instances implementing a statement a *functional unit* (FU), and we aim to select how many instances of each functional unit should be implemented in the accelerator. We note that depending on the kernels being merged, syntactically identical statements (after variable renaming) may occur: in this case two functional units may compute exactly the same operations, albeit perhaps with different iteration domains. This can be easily detected from the kernels representations, and we merge into a single FU those computing identical operations, to facilitate solving the optimization problem below. Note in our simple compilation phase to convert the input straight-line program to a sequence of instructions, we exploit the fact that multiple kernels/functionalities may be mapped to the same FU, perhaps by adjusting the loop bounds passed as argument to the instruction, to find a compact scheduling+placement.

Number of replications The key challenge is to determine the number of replications of each kernel/functionality, given that (a) different workloads may expose vastly different amount of parallelism; and (b) how many elementary operation(s) can be shared between kernels is related to the number of replications of each kernel. Our objective is to optimize throughput per area, in other words, we aim to increase resource sharing without performance penalty, something typically achievable when operations would anyway be otherwise idling due to sequences of dependent kernel calls.

For a particular workload summarized as the number of calls to each kernel, we aim to minimize the expected execution time under resource constraints, summarized in the following optimization problem:

$$\text{minimize} \quad \sum_{K\in Kernels} \lceil \#calls(K)/\#FU(K) \rceil * card(\mathcal{D}_K) * IL_K$$

$$\text{subject to} \quad \sum_{i\in FUs} Area(FU_i) * \#FU_i < \text{max\_area}$$

Where a FU, or Functional Unit, is a *hardware implementation* of the operations in a kernel K, and IL_K the iteration latency to execute one inner-most loop iteration, that is the latency of the FU to execute once. The unknown to be computed is the number of FU, for each FU type. The workload mix, given by the number of calls to each kernel/functionality, is input to this optimization problem. As we weight the latency of an iteration by the cardinality of its iteration domain, in case of an heterogeneous workload combining N^2 (e.g., matrix addition) and N^3 (e.g., matrix-multiplication) operations, the dominant cost driving the solution found will be for the N^3 operations. $Area(FU)$ is computed by approximating the DSP consumption of an FU, itself adjusted if operations in a FU can be shared across multiple FUs: they are of the same type, and can execute in pipelined fashion. In practice, to solve this problem we simply enumerate all solutions (i.e., number of FUs of each type) and for each compute latency and resources. We output the first solution that meets resource constraints with minimal latency.

4 Accelerator Implementation

In this section, we analyse the modules that compose the accelerator and discuss their implementation.

Structure of the Accelerator The accelerator layout is illustrated Fig 5, and is orchestrated around a single pipelined loop dispatching user-specified computing tasks to Functional Units (FUs), corresponding to loop bodies of merged kernels. Such tasks are composed of four stages: the computation of the read/write locations as function of the current loop iterator, the loading of the data, the computation, the storage. The accelerator may be broken down to three submodules: the loop control logic, handling dispatch of the operation to the FU, the FUs themselves and the local buffer storing the required data.

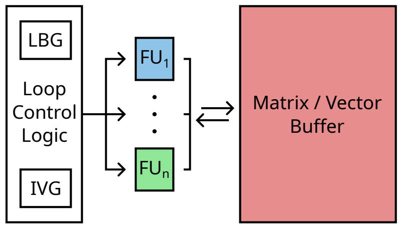

Figure 5. Layout of the Generic Accelerator

While our accelerator architecture is designed for Xilinx Ultrascale+ MPSoC [29], that is, FPGA integrated with a CPU, none of its features depends on CPUs. Therefore, some implementation details may be specific such as the communication-handling logic and the wrapping application mentioned below, but they do not limit the genericity of our approach.

Iteration Vector Generator (IVG) Though merged kernel are transformed to iterate on the same space, additional logic is still needed to convert the global (scalar) loop index to the iteration vector given as input to the FU – typically indexes of accessed arrays. This computation is done by the IVG, that initialise the iteration vectors to 0_d and update them using their former value through a fixed state machine.

Functional Units Functional Units (FU) are specialized, fully pipeline units capable of executing one or more operations of the input kernel such as addition, multiplication, data movement (e.g. for transposition), etc. They take as input the iteration vector, load directly values from the local buffer depending on the current iteration vector value, and compute in one or more outputs, which are directly stored back to the local buffer. An FU can expose several computation kernels capabilities, e.g. matrix addition and subtraction, but cannot operates in our implementation on more that 2 source arrays and 1 destination array. This restriction is lead by the gubernatorial amount of logic needed to load arbitrary values from the local buffer from HLS-based designs, as discussed in Sec. 7.

In this case, we allow further sharing by flipping the sign bit of one of the FP16-encoded operand, then using the adder to perform $a + (-b)$ with no additional DSP cost.

The load and store locations are constrained by the local buffer implementation detailed below: due to memory port pressure, FUs can only operate on a non-shared pool of the local buffer during the time of their kernel computation, but the data is afterward accessible by all FUs. As FUs are fully pipelined, they must have no loop-carried dependence: the minimal reuse distance of read-after-write dependence has to be higher than the latency of the complete FU.

Loop Bound Generator (LBG) After each end of execution of kernels, the LBG scans all next kernels and compute the cardinal of the minimal iteration space by maxing their iteration space sizes. This value is then used as the trip count of the FU-scheduling loop.

Loop Control Logic The accelerator is organised around a single loop defined in HLS-C++ as a `for` ranging from 0 to a maximum value given by the LBG. This loop corresponds to a flattened version of the fully merged loops of all accelerated kernels, and is pipelined to achieve a maximum throughput of 1 execution of all FU per cycle, i.e. to fully exploit all FUs.

The role of the loop control logic is twofold. First, it schedules operations on the FU, and second it iterates over all merged kernels to ensure a correct and complete execution of the input workload.

Off-Chip Communications Data in the local buffer are coalesced [32] for transfers into 64-bit packets that are sent or received together in one burst to/from the off-chip DRAM before and after execution of the accelerator. Execution time is measured by an on-chip counter wired to the main clock, whose value is fetched before the execution of the computation and right after its termination (i.e. not including communications). The local buffer communicates with off-chip memory using the AXI4 bus connected to one high-performance communication port of the Zync MPSoC. Its setup is controlled by MMIO-mapped registers using an AXILite bus, managed by a wrapper C++ application running on CPU integrated in the ZCU104 MPSoC. This application also handles the execution flow as well as memory management from an embedded Linux OS, using libraries provided by the PYNQ framework as well as autogenerated drivers from Vitis HLS.

Access to the Local Buffer Each FU loads and stores data from a global, on-chip buffer implemented with double-port BRAMs whose accesses are spread over a three-stage pipeline: read, execute and write in order to conserve the initiation interval of 1 of the FU enforced by the loop control logic. To allow off-chip communications at a rate of 64-bit per cycle, the Local Buffer is partitioned cyclically by a factor of 2, allowing 4 simultaneous FP16 loads/stores.

5 Experimental Results

In this section, we will analyse the performance of two merged accelerators whose characteristics are reported in Tbl. 3: one optimized for dense linear algebra computation, the other for the computation of correlation matrices, as expressed in PolyBench/C [25]. All measurements are done on a ZCU104 board running the PYNQ 2.6 Linux image, and all IP are generated from annotated C++ code using Xilinx Vitis 2022.1 [30] on Linux 6.0.7. Resource usage is measured after out-of-context P&R for each HLS accelerator, which excludes AXI data routing to the integrated CPU. Cycles measurements of the proposed accelerator are taken from on-chip counter on the target board, whereas custom accelerators execution time are computed from the pipeline latency given by the HLS Tool report. Unless specified, the data type used is 16-bit floating point. The total functionalities of the accelerators are summarized in Tbl. 4.

Due to our implementation, we allow every merged microkernel to be executed on a compatible FU. Therefore, the only difference of FU between the Linear Algebra (LA-GA) and Correlation (CORR-GA) rely in 1) the support of triangular iteration space for GA-LA and 2) the presence of one square root / division unit for GA-CORR. On all accelerators, only three different types of FU are integrated:

- one capable of handling `mulmm` and all the `mul` and `add`/`sub` derivatives, based on two operators ($\pm$ and $*$)
- one only handling `add` and its derivatives (including `sub`) composed of one operator $\pm$
- one handling `sqrt` and `div`, based on two operators: $\sqrt{\cdot}$ and /

	Number of operators				Nb. of	IVG supports	Local Buffer
	$a \pm b$	$a * b$	a/b	$\sqrt{a}$	FU	triangular loops	Size
BLAS	2	1	0	0	2	Yes	25 Matrices
CORR	3	3	1	1	4	No	25 Matrices

Table 3. Configuration of the LA-GA accelerator and the Correlation accelerator

Kernel	Description	Op.	LA-GA	CORR-GA
`noop`	Do nothing	None	✓	✓
`mulmm`	Matrix-matrix multiplication	$\pm$ and $*$	✓	✓
`mulmv`	Matrix-vector multiplication	$\pm$ and $*$	✓	✓
`multrmm`	Triangular matrix-matrix multiplication	$\pm$ and $*$	✓	
`multrmv`	Triangular matrix-vector multiplication	$\pm$ and $*$	✓	
`mulsm`	Scalar-matrix multiplication	$*$	✓	✓
`multrsm`	Scalar-triangular matrix multiplication	$*$	✓	
`mulsv`	Scalar-vector multiplication	$*$	✓	✓
`muls`	Scalar-scalar multiplication	$*$	✓	✓
`trm`	Matrix transposition	None	✓	✓
`addm`	Matrix addition	$\pm$	✓	✓
`addv`	Vector addition	$\pm$	✓	✓
`adds`	Scalar addition	$\pm$	✓	✓
`addtrm`	Triangular matrix addition	$\pm$	✓	
`subm`	Matrix subtraction	$\pm$	✓	✓
`subcmv`	Column-wise matrix subtraction	$\pm$	✓	✓
`subv`	Vector subtraction	$\pm$	✓	✓
`subs`	Scalar subtraction	$\pm$	✓	✓
`pmulm`	Point-wise matrix multiplication	$*$	✓	✓
`pmulv`	Point-wise vector multiplication	$*$	✓	✓
`oprodv`	Outer (vector) product	$*$	✓	✓
`sqrtv`	Point-wise vector square root	$\sqrt{\cdot}$		✓
`sqrts`	Scalar square root	$\sqrt{\cdot}$		✓
`accsumcm`	Columns-wise accumulation of a matrix	$\pm$	✓	✓
`cutminv`	Vector round to 1 low values	None	✓	✓
`divms`	Pointwise division of matrices	/		✓
`divvs`	Pointwise division of vectors	/		✓
`divcmv`	Point-wise division with column-wise value	/		✓
`set0m`	Initialisation of a matrix to 0	None	✓	✓
`setidm`	Initialisation of a matrix to *Id*	None	✓	✓
`setd1`	Initialisation of the diagonal of a matrix to 1	None	✓	✓

Table 4. Supported kernel by either the Correlation or the Linear Algebra accelerator

We compare our accelerator with the Max Sharing (MS) design, where only one physical operator accelerator for each operation type is instantiated, and the Max Throughput (MT) that achieves minimal execution time while keeping all data in a local buffer of the same characteristics than the generic accelerator one. *On both MT and MS, no genericity of the design is possible*, i.e. only a single benchmark can be executed. We evaluate our generic accelerator on three metrics: execution time (in cycle), throughput per area, computed as $\frac{NB\_FLOP}{EXEC\_TIME*NB\_RESOURCE}$ with NB_FLOP the number of floating-point operation in the input benchmarks, and $MN\_RESOURCE$ the number of DSP or chunk of 10 000 FF / LUT in the design; and occupancy, computed as $\frac{NB\_OP}{EXECUTION\_TIME*NB\_UNIT}$, with NB_OP the number of operations being executable by the operator in the input program, and NB_UNIT the number of compatible units in the design.

5.1 Linear Algebra

Execution time, resources and performance per area metric are reported in Tbl. 5, while Tbl. 6 report occupancy. Performances on batches of 5 independent problems are also evaluated, in Tbl. 7. The accelerator for linear algebra, noted LA-GA is composed of two FUs: (a) one which supports matrix multiplication, thus integrating an adder and a multiplier, as well as all kernels relying on either an addition, a multiplication, a subtraction or a transposition; and (b) one which only supports transposition as well a additions/subtractions.

This choice was guided by the necessity to cover all linear algebra expressions (hence the first "generic" FU) while being able to coarse-grain pipeline kernels composed of both multiplication and additions such as SCALE.

Indeed, SCALE is composed of two kernels: the first one is `mulsm` (multiplication scalar-matrix, with $O(N^2)$) and the second one is `addm` (addition of matrix, also with $O(N^2)$ complexity). The accelerator is then able to execute in parallel two different instances of SCALE on its two FUs. However, this does not achieve full usage of the compute units as the adder, also present in the first FU, stays idle. Moreover, the coarse-grain pipeline must be filed and emptied at the start of the batched execution sequence, which limits its maximum occupancy to $1 - \frac{2}{BATCH\_SIZE+2}$ for a pipeline composed of 2 stages, resulting in, 71 % for a batch size of 5.

On SCALE and GER, the LA-GA is around 2 times slower than MS for non-batched workloads. This is due to the dedicated accelerator expressing in one fully pipelined loop nest the complete application, whereas the LA-GA splits it in several (fully pipelined) kernels, increasing the final latency. These differences fade away when the input is batched as the LA-GA will overlap kernel execution through coarse-grain pipelining and preset systematically faster execution that MS except for GER, where the batching factor is not enough to benefit from coarse-grain pipelining due to a 3-stage pipeline.

However, occupancy and throughput-per-area falls behind both MS and MT for two reasons: first, coarse-grain pipelining is limited by the first and last stages of the pipeline; and the genericity of the accelerator is achieved at the cost of area, both because of glue logic and idling units. The former can be quantified by the performance-per-LUT and performance-per-FF, which remains between 3 and 20 times lower than dedicated accelerator. For idling units, we take the GER BLAS primitive as an example. GER does not use `mulmm` no `mulmv` (or their triangular derivative), which means that a maximum of one unit (either $\pm$ or $*$) is active in the first FU. Conversely, dedicated accelerators (both MS or MT) do not have this constraint, hence resulting in higher occupancy.

5.2 Correlation

For data science applications such as Covariance, linear algebra primitives are not sufficient as other kernels are needed: column-wise accumulation of the matrix (a one-kernel implementation of $A^t \cdot 1_{vector}$), column-wise subtraction of a vector to a matrix and cut-of of a vector (used to avoid floating-points error when divising by a near-zero value) as well as division and square root.

Therefore, we enriched our accelerator with one FU merging these two kernels to create the CORR-GA accelerator whose configuration is detailed in Tbl. 3: (a) 3 FU capable of computing `mulmm` and all derivatives (kernels relying on $\pm$ and/or $*$); and (b) 1 FU capable of computing either $\sqrt{\cdot}$ or /

This topology was tailored to a batched execution of size 3: as seen in Sec. 2, $*$ and / are shareable instructions across batches as they are dominated in terms of occupancy time by the final matrix multiply of the correlation computation. Reports of the execution time as well as performance-per-area metrics are summarized in Tbl. 8, while occupancy of the units is detailed in Tbl. 9.

Globally, the CORR-GA performs similarly to the LA-GA: execution time is slower than MS due to the decomposition of kernels that are otherwise expressible in one loop nest. However, as the accelerator was tailored for 3 executions of CORR (instead of a trade-off of all benchmarks), we exhibit occupancy gains in this workload. Indeed, occupancy of $\pm$ and $*$ units are dominated by the final matrix multiplication of CORR, while $\sqrt{\cdot}$ and / usage tripled due to their sharing between the three batched instance.

5.3 Scaling and comparison

We evaluate the scalability of our approach on three different aspect: data type, number of entries of the local buffer and problem size. Area measurements are reported after P&R in Tbl. 10. While switching from half precision to double precision doubles LUT due to the additional routing resources necessary to handle the supplementary data, the accelerator only increase by around 25 % when quadrupling the size of the local buffer. This is due to the fact that loading and storing units that are the only elements to scale with its size: the remaining data dispatch, FU selection and iteration vector generation stay identical. On the other hand, LUT and FF and DSP usage increase linearly with the number of FU, suggesting that our approach does not generate quadratic amount of logic with respect to its raw computation power. However, synthesis time increases significantly with the number of FUs, reaching several hours for a GA with 10 FUs.

Bench	Arithmetic	Execution Time (cycles)			FLOP/C/DSP			FLOP/C/10kFF			FLOP/C/10kLUT		
name	expression	MS	MT	LA-GA	MS	MT	LA-GA	MS	MT	LA-GA	MS	MT	LA-GA
SCALE	$A = \alpha \cdot A + B$	5572	2059	8258	0.368	0.497	0.165	5.080	13.593	2.193	7.722	20.466	1.053
GEMV	$y = \alpha \cdot A \cdot x + \beta \cdot y$	4553	2126	4396	0.457	0.391	0.315	3.960	12.950	4.184	5.686	18.752	2.010
TRMV	$y = \alpha \cdot A \cdot x + \beta \cdot y$	2339	2435	2380	0.458	0.293	0.300	6.177	5.894	3.982	9.311	8.735	1.913
GER	$A = \alpha \cdot x \cdot y^t + A$	4738	2057	8343	0.436	0.401	0.165	6.093	13.528	2.187	9.348	20.058	1.051
GEMM	$C = \alpha \cdot A \cdot B + \beta \cdot C$	307586	134018	274540	0.433	0.397	0.323	5.759	12.934	4.287	8.860	19.011	2.059
TRMM	$C = \alpha \cdot A \cdot B + \beta \cdot C$	149696	155840	145516	0.458	0.293	0.314	5.964	5.816	4.169	8.991	8.688	2.002

Table 5. Throughput of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator (LA-GA) for several Linear Algebra Benchmarks

Bench	Occupancy (±)			Occupancy (∗)			Global Occupancy		
name	MS	MT	LA-GA	MS	MT	LA-GA	MS	MT	LA-GA
SCALE	73.51%	99.47%	24.80%	73.51%	99.47%	49.60%	73.51%	99.47%	41.33%
GEMV	89.96%	96.33%	46.59%	92.77%	49.67%	96.09%	91.37%	65.22%	79.59%
TRMV	88.93%	85.42%	43.70%	94.40%	45.34%	92.77%	91.66%	58.70%	76.41%
GER	86.45%	99.56%	24.55%	87.80%	50.56%	49.86%	87.13%	66.89%	41.42%
GEMM	85.23%	97.80%	47.74%	87.89%	67.24%	98.47%	86.56%	79.46%	81.56%
TRMM	88.93%	85.42%	45.74%	94.40%	90.68%	97.11%	91.66%	88.05%	79.99%

Table 6. Occupancy of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator (LA-GA) for several Linear Algebra Benchmarks

Bench	Exec. Time (cycles)			Occupancy ($a \pm b$)			Occupancy ($a \cdot b$)			Global Occupancy		
name	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA
SCALEx5	27860	10295	24726	73.51%	99.47%	41.41%	73.51%	99.47%	82.83%	73.51%	99.47%	69.02%
GEMVx5	22765	10630	21544	89.96%	96.33%	47.53%	92.77%	49.67%	98.03%	91.37%	73.00%	81.20%
TRMVx5	11695	12175	11379	88.93%	85.42%	45.70%	94.40%	45.34%	97.02%	91.66%	65.38%	79.91%
GERx5	23690	10285	41279	86.45%	99.56%	40.71%	87.80%	50.56%	82.70%	87.13%	75.06%	68.71%
GEMMx5	1537930	670090	1356136	85.23%	97.80%	48.33%	87.89%	67.24%	99.67%	86.56%	82.52%	82.56%
TRMMx5	748480	779200	711016	88.93%	85.42%	46.81%	94.40%	90.68%	99.37%	91.66%	88.05%	81.85%

Table 7. Occupancy of a custom IP optimised for maximum efficiency and the Custom Generic Accelerator (LA-GA) for a batched subset of Linear Algebra Benchmarks

Bench	Arithmetic	Execution Time (cycles)			FLOP/C/DSP			FLOP/C/10kFF			FLOP/C/10kLUT		
name	expression	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA
CENTER	$X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$	8343	4166	12480	0.495	0.495	0.055	10.362	19.448	1.090	9.570	15.374	0.425
STDDEV	$\sigma^X_j = \sqrt{\sum_i (X^C_i)^2/n}$	16691	8370	29053	0.247	0.247	0.047	7.796	13.991	0.936	6.148	10.148	0.365
CENTER-REDUCE-DIV	$X^{CR}_{ij} = \left(X_{ij} - \sum_{i'} X_{i'j}\right)/(\sigma^X_j \cdot \sqrt{n})$	20935	10486	33352	0.247	0.164	0.052	6.579	9.707	1.021	5.579	7.761	0.398
CORR	$(X^{CR})^t \cdot X^{CR}$	291221	144614	303763	0.468	0.314	0.150	10.905	17.962	2.955	9.119	13.834	1.152
CORRx3	3×CORR	873663	433842	320603	0.468	0.314	0.425	10.905	17.962	8.400	9.119	13.834	3.275

Table 8. Throughput of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator (CORR-GA) for Covariance subexpressions

Bench	Occupancy (±)			Occupancy (∗)			Occupancy (/)			Occupancy ($\sqrt{\cdot}$)			Global occupancy		
name	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA	MS	MT	CORR-GA
CORR	94.23%	94.88%	30.11%	91.44%	46.04%	29.22%	0.02%	0.02%	0.02%	1.43%	1.44%	1.37%	46.78%	37.68%	22.43%
CORRx3	94.23%	94.88%	85.60%	91.44%	46.04%	83.06%	0.02%	0.02%	0.06%	1.43%	1.44%	3.89%	46.78%	37.68%	63.74%

Table 9. Occupancy of a custom IP custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator (CORR-GA) for Correlation subexpressions

Data Type	Nb. Entries	Nb. FU	Pb. Size	LUT	FF	DSP	BRAM
FP16	25	2	64	9418	4524	6	109
FP64	25	2	64	19535	7428	14	109
FP16	**50**	2	64	10945	4705	6	109
FP16	**100**	2	64	11899	4952	6	409
FP16	25	2	**150**	12701	4939	8	650
FP16	25	**4**	64	14723	5971	12	109
FP16	25	**6**	64	20957	7382	18	109
FP16	25	**10**	64	31998	10947	30	112
FP16	25	**20**	64	HLS Synthesis time out (> 3h)			

Table 10. Scaling properties of the LA-GA accelerator

Data Type	Implementation	OP/Cycle/DSP
INT32	ResNet-18 ScaleHLS [33]	1.343
INT32	ResNet-18 TVM-VTA [23]	0.344
INT32	LA-GA GEMM	0.646
FP32	GEMM ScaleHLS	0.393
FP32	LA-GA GEMM	0.277

Table 11. Performance per area comparison with data extracted from other published accelerators

We also provide as a indicative example in Tbl. 11 a comparison of our performance against two state-of-the art designs dedicated to machine learning workloads: ScaleHLS [33] and VTA [23], on GEMM, extrapolated from their FP16 (2 DSP per addition, 2 DSP per multiplication) to an FP32 projection (2 DSP per addition, 3 DSP per multiplication) and and INT32 one (0 DSP per addition, 3 DSP per multiplication) from the ScaleHLS publication, and compare to ours. On this metric, we achieve comparable performance to their designs, however ScaleHLS optimizes a single workload and is not producing a generic accelerator.

6 Related Work

The topic of semi-specialized accelerator design [5, 18, 20] have been widely studied, targeting a variety of subdomains such as encryption [21], graph processing [4] or machine learning [1, 3, 12, 34]. For example Cong et al. [8] propose a technique to quickly generate accelerators on a template architecture, but targets single application acceleration on MPSoC, in contrast to our multi-functionality approach. ScaleHLS [33] is an end-to-end MLIR-based framework for throughput-optimized accelerator generation, and does not target merging efficiently multiple functionalities/kernels to accelerate.

The Versatile Tensor Accelerator [23] relies on concepts of decomposition of programs into kernels for deported execution on an accelerator, but is optimised for deep learning with functional units limited to configurable GEMM and tensor-specialised ALU, for which kernels have to be described using micro-code instead of directly integrating them in the design. In contrast, the functional units themselves that are candidate for resource sharing are design parameters in our generic accelerator design.

Resource sharing for area-efficient accelerator generation is another weel-studied research topic [16, 26]. Li et al.[17] proposed a method based on loop body components requirements to create area-efficient design suited for coarse-grain replication. However, this method does not consider neither coarse-grain pipelining nor generality of the accelerator. Jain et al. [15] manually derive the accelerator architecture from targeted worklaods, while we automatically infer its parameters given polyhedral description of the functionalities.

Morvan et al. [24] tackled the problem of under-usage of imperfectly nested loop pipelining by automated insertion of padding computations. Such an approach can be applied on the merged loops presented in this paper to further reduce idle time of the FU.

7 Limitations

Though the kernel merging approach for general accelerator generation is promising, our implementation suffers from several flaws, both on the technical side (unused/overused FPGA resources) and on our evaluation of the accelerator.

Routing between FU and Buffers Our implementation allows every FU can access every memory location of the local buffer for easier customisation of the generated accelerator. Indeed, a generic local buffer load/store IP is integrated for every FU, that rely on costly multiplexers, which can be avoided by specializing it to the access pattern of the FU.

Deeper polyhedral analysis and re-scheduling may also exhibit cross-FU reuse when the same data is used by 2 different FU. A future research direction may be to ensure maximal merging of these data path to avoid as best as possible redundant loading; but we expect this analysis to lead to few real-life usse cases.

Merging of Kernels with Different Iteration Space In all tested benchmarks, the iteration space vector can be shared amongst all merged kernel. However, this is not true in general: two kernels may iterate over dimensions of different size, which requires the generation of two iteration vector by the IVG. This leads to additional LUT-based logic limiting the application of our approach on LUT-constraints designs, but should not disturb the execution time

Data Reuse: Optimizing Buffer Communication Our implementation does not consider reuse of data inter iteration of the FU, as this may introduce loop carried dependencies and thus stall the pipeline. However, short-distance single-producer / multiple consumer data can be kept in FF-based memory to alleviate BRAM's load, diminishing pressure on ports and allowing further parallel computation on the now-loadable data.

Further Sharing of Operations inside FU Our current framework does not allow an FU realising a fused multiply-add (FMA) to work as a both an adder and a multiplier at the same time. However, such a behaviour is crucial for maximal usage of the accelerator on tasks graphs that do not rely on FMA, but still uses both additions and multiplications as (single) operators. Adding such capabilities require a significant increase of the front-end on the FU, in the sense that additional logic and routing is needed for the transmission of second operator arguments; logic that cannot be reused in any non-parallel kernel. Therefore, we expect a trade-off between the occupancy of the operators and the amount of glue needed for the flexibility to fully utilise them.

Vectorisation of the FU In our evaluation, we only consider FU composed of one operation of fused multiply-add. While this is a technical limitation of our current implementation, there is no reason to do so in the general case. Though we expect the relative quantity of glue logic to decrease with the size of the FUs, we also expect sub-optimal uses of these larger FUs as they also come with a more specialised (hence less reusable) operation graph that single or double-operation units.

8 Conclusion

While the accessibility and ease-of-use of fixed-function accelerator design has significantly increased, updating the functionalities being accelerated can be tedious if at all possible. In contrast, multi-purpose accelerators aim to keep most benefits of fixed-function acceleration, while being efficient on a variety of workloads.

In this work, we presented a system to build a throughput-oriented, multi-functionality accelerator from a set of input polyhedral kernels. We conducted detailed evaluation with on-board measurements of two generic accelerators for dense linear algebra workloads, exposing the merits and limitations of such multi-functionality accelerator design.

References

[1] Stefan Abi-Karam, Yuqi He, Rishov Sarkar, Lakshmi Sathidevi, Zihang Qiao, and Cong Hao. 2022. GenGNN: A Generic FPGA Framework for Graph Neural Network Acceleration. *CoRR* abs/2201.08475 (2022). arXiv:2201.08475 https://arxiv.org/abs/2201.08475

[2] C. Bastoul. 2004. Code Generation in the Polyhedral Model Is Easier Than You Think. In *13th International Conference on Parallel Architectures and Compilation Techniques, PACT*. IEEE, Antibes, France, 7–16.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799* 11, 2018 (2018), 20.

[4] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. 2021. ThunderGP: HLS-Based Graph Processing Framework on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 69–80. https://doi.org/10.1145/3431920.3439290

[5] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. 2012. CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design* (Redondo Beach, California, USA) *(ISLPED '12)*. Association for Computing Machinery, New York, NY, USA, 379–384. https://doi.org/10.1145/2333660.2333747

[6] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. 2016. Software infrastructure for enabling FPGA-based accelerations in data centers. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. 154–155.

[7] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[8] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. 2018. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465940

[9] Paul Feautrier. 1991. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20, 1 (1991), 23–53.

[10] P. Feautrier. 1992. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming* 21, 6 (1992), 389–420.

[11] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations. *International Journal of Parallel Programming* 34, 3 (June 2006), 261–317.

[12] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. https://doi.org/10.1109/FCCM.2017.25

[13] Andrei Hagiescu, Weng-Fai Wong, David F Bacon, and Rodric Rabbah. 2009. A computing origami: Folding streams in FPGAs. In *Proceedings of the 46th Annual Design Automation Conference*. 282–287.

[14] Intel. 2022. High Level Synthesis Compiler. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[15] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. 2016. Throughput oriented FPGA overlays using DSP blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1628–1633.

[16] Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2022. Resource Sharing in Dataflow Circuits. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 1–9.

[17] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. 2015. Resource-Aware Throughput Optimization for High-Level Synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, California, USA) *(FPGA '15)*. Association for Computing Machinery, New York, NY, USA, 200–209. https://doi.org/10.1145/2684746.2689065

[18] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. 2015. QuickDough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. In *2015 International Conference on Field Programmable Technology (FPT)*. 56–63. https://doi.org/10.1109/FPT.2015.7393130

[19] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, et al. 2022. OverGen: Improving FPGA Usability through Domain-specific Overlay Generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 35–56.

[20] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. 2018. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 245–257. https://doi.org/10.1109/MICRO.2018.00028

[21] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki. 2017. HLS design of a hardware accelerator for Homomorphic Encryption. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 178–183. https://doi.org/10.1109/DDECS.2017.7934578

[22] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. 2019. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro* 39, 5 (2019), 8–16.

[23] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware–Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16. https://doi.org/10.1109/MM.2019.2928962

[24] Antoine Morvan, Steven Derrien, and Patrice Quinton. 2013. Polyhedral Bubble Insertion: A Method to Improve Nested Loop Pipelining for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 3 (2013), 339–352. https://doi.org/10.1109/TCAD.2012.2228270

[25] L.-N. Pouchet. 2011. PolyBench: The Polyhedral Benchmarking suite, version PolyBench/C 4.2.1. http://polybench.sf.net. Last accessed: May 2017.

[26] Bajaj Ronak and Suhaib A Fahmy. 2016. Multipumping flexible DSP blocks for resource reduction on Xilinx FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 9 (2016),

1471–1482.

[27] Nicholas Weaver. 2008. Retiming, repipelining and c-slow retiming. *Reconfigurable Computing* (2008), 383–399.

[28] Xilinx. 2022. The Merlin compiler. https://github.com/Xilinx/merlin-compiler.

[29] Xilinx. 2022. *UltraScale Architecture Configuration User Guide (UG570)*.

[30] Xilinx. 2022. *Vitis High-Level Synthesis User Guide (UG1399)*.

[31] Xilinx. 2022. Vitis Unified Software Platform. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

[32] Xilinx. 2022. *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*.

[33] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. 2022. ScaleHLS: A New Scalable High-Level Synthesis Framework on Multi-Level Intermediate Representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 741–755.

[34] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. 2020. Achieving Full Parallelism in LSTM via a Unified Accelerator Design. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*. 469–477. https://doi.org/10.1109/ICCD50377.2020.00086

[35] Heidi Ziegler, Byoungro So, Mary Hall, and Pedro C Diniz. 2002. Coarse-grain pipelining on multiple FPGA architectures. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 77–86.