Journal of Signal Processing Systems (2023) 95:1181–1201
https://doi.org/10.1007/s11265-023-01885-5

The Good, the Bad and the Ugly: Practices and Perspectives on Hardware Acceleration for Embedded Image Processing

Joshua Fryer[1] · Paulo Garcia[2]

Received: 28 November 2022 / Revised: 14 July 2023 / Accepted: 18 July 2023 / Published online: 29 July 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

Modern embedded image processing deployment systems are heterogeneous combinations of general-purpose and specialized processors, custom ASIC accelerators and bespoke hardware accelerators. This paper offers a primer on hardware acceleration of image processing, focusing on embedded, real-time applications. We then survey the landscape of High Level Synthesis technologies that are amenable to the domain, as well as new-generation Hardware Description Languages, and present our ongoing work on IMP-lang, a language for early stage design of heterogeneous image processing systems. We show that hardware acceleration is not just a process of converting a piece of computation into an equivalent hardware system: that naive approach offers, in most cases, little benefit. Instead, acceleration must take into account how data is streamed throughout the system, and optimize that streaming accordingly. We show that the choice of tooling plays an important role in the results of acceleration. Different tools, in function of the underlying language paradigm, produce wildly different results across performance, size, and power consumption metrics. Finally, we show that bringing heterogeneous considerations to the language level offers significant advantages to early design estimation, allowing designers to partition their algorithms more efficiently, iterating towards a convergent design that can then be implemented across heterogeneous elements accordingly.

Keywords Image processing · Embedded · FPGAs · Hardware acceleration · Language · Paradigm · Co-design

1 Introduction

Image processing was once delegated to offline activities [1]: image restoration, improvement, augmentation [2]; diagnostics or information retrieval from visuals [3]; manipulation for creative purposes [4]; etc. In the last two decades, as computing power grew (whilst form factors and costs decreased [5]), image processing was brought to the forefront of real-time processing [6]. More generally, computer vision (where "vision" may include input from cameras, radars, LiDars, etc. [7]) now powers myriad application; from safety and security (tracking, behavior analysis [8]) to autonomous robots (scene perception [9]), and many others in between [10].

✉ Paulo Garcia
Paulo.G@chula.ac.th

Joshua Fryer
josh.s.fryer@gmail.com

[1] Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada

[2] International School of Engineering, Chulalongkorn University, Bangkok, Thailand

Of course, this increase in computing power creates a positive feedback loop: designers create more and more complex image processing algorithms (handcrafted [11], or, more recently, implemented through statistical training methods through Machine Learning [12]) which in turn fuel the need for faster, cheaper, more energy-efficient, deployment platforms. The time when this could be achieved by general-purpose processors alone is long gone. Heterogeneity is the name of the game.

Modern image processing deployment systems are heterogeneous combinations of general-purpose and specialized processors (e.g., DSPs [13]), custom ASIC accelerators (e.g., GPUs [14]) and bespoke hardware accelerators (deployed, e.g., on FPGAs [15]). This complexity is required to support the desired levels of performance, power consumption [16], determinism [17], and form-factor: i.e., through a design process that must partition an algorithmic description appropriately [18], choose the best architectural artifact for executing each partition, implementing the desired computation using hardware/software methodologies [19] and toolchains [20], and integrating and interoperating all the moving parts into a coherent system.

Unsurprisingly, this is not easy. As the level of complexity grows, so does the need for automated tools and methods that facilitate design. In this article, we concern ourselves with a small aspect of this ecosystem: hardware acceleration for embedded image processing [21], i.e., the process of creating bespoke accelerators on FPGA logic. We begin by offering a primer on hardware acceleration, showing how FPGA acceleration should (and should not) be used, in the context of embedded (real-time) image processing. We then offer a review of the extant tools (specifically: High Level Synthesis (HLS) from different source languages [22]) for this purpose, with quantitative examples that demonstrate how different tools provide wildly different results, that the keen designer must be aware of. We conclude by describing our ongoing work on an open-source design language purposely built for the early design space exploration of heterogeneous embedded image processing systems.

Specifically, this article offers the following contributions:

- We provide a primer on hardware acceleration of image processing, useful for practitioners entering the field (Sect. 2).
- We review the state of High Level Synthesis technology, focusing on the different paradigms utilized across languages, and how these affect generated designs (Sect. 3).
- We introduce IMP-lang, a language for heterogeneous design space exploration, amenable to the deployment of modern embedded image processing systems (Sect. 4), and show how it can be used for early design evaluation (Sect. 5).

Section 6 concludes this article, offering perspectives on ongoing efforts in this field.

2 Background: FPGAs and Acceleration Strategies

Hardware acceleration is the process of decreasing the total execution time (performance-oriented acceleration) or decreasing the total power consumption (power-oriented acceleration) of a given application (i.e., a software task) by moving parts of that task to dedicated hardware.

In this manuscript, we'll strive to describe how hardware acceleration works, in function of architecture and specific task behavior, as well as identify common misconceptions about hardware acceleration to prevent designers from falling into these pitfalls. The examples in this manuscript assume the original task to be accelerated is running on a single-core system for simplicity, but the lessons apply equally to multi-core systems.

We are assuming that acceleration is performed on Field Programmable Gate Arrays (FPGA): for those inexperienced with them, there are three properties that must be understood to fully comprehend the rest of this manuscript: I/O interfaces, clock frequencies, and internal memory.

2.1 I/O Interfaces

A processor typically has a single I/O interface: memory ports. Memory ports are a collection of I/O pins that implement an address-based read/write interface. Typically, this interface is connected to a bus, where memory and other peripheral devices are attached to. Depending on the address, the bus system will propagate requests to the corresponding device; either main memory, or a peripheral device such as a network card. All devices that can initiate memory accesses compete for bus use, and adding a new device that can transfer data to the processor must be done by appending it to the bus.

FPGAs, on the other hand, possess far more available I/O pins, and, because of their configurable nature, there is no hard-coded behavior for each pin. Thus, it is possible to connect several devices directly to an FPGA (without an intermediate bus), and internal logic (operating in truly parallel fashion) can transfer data to/from several of those devices at the same time. It is, of course, still possible to connect an FPGA to a bus.

2.2 Clock Frequencies

The highest achievable clock frequency in any digital circuit is a function of the propagation delays (read: number of logic gates in series) in that circuit. Because FPGAs are comprised of programmable logic, there is always an overhead in the number of gates in series, for a given circuit, compared to the same circuit implemented in Application Specific Integrated Circuit (ASIC). This means that, for an FPGA and an ASIC fabricated using the same technology implementing the same circuit, the FPGA always yields a lower clock frequency. Any digital circuit can be implemented on FPGA, including a complete processor: but, it will *always* have worse performance than the same processor implemented on ASIC.

2.3 Internal Memory

FPGAs consist of configurable logic, capable of implementing any computation, and distributed memory blocks. These distributed memory blocks are typically small (a few kilobytes), and their interconnection is controlled by configurable logic. Thus, it is possible to implement large, monolithic memory structures, similar to external main memories. But, more importantly, it is possible to implement custom memory architectures, with computations in-between storage. This feature is one of the most powerful capabilities

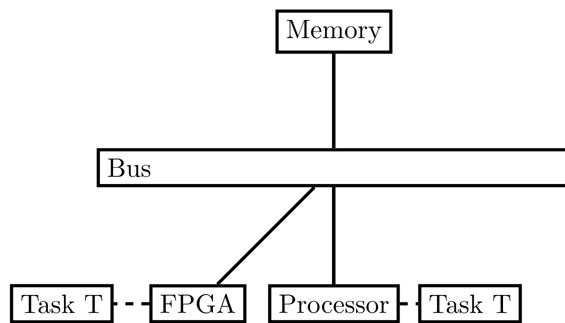

Figure 1 Processor-memory and FPGA-memory interconnect through shared bus. This architecture applies the prototypical processor-memory connection to FPGA design, failing to take advantage of bespoke memory systems.

of FPGAs: we'll see how this can be used in practice in the examples below.

2.4 Performance Bounds

In this section we'll consider completely converting a software task into an equivalent hardware one; i.e., such that an FPGA completely replaces a processor in a system. In the next section, we'll consider accelerating parts of a software task.

Consider the system depicted in Fig. 1: a single-core processor, connected to a memory via a bus, executing a task *T*. This system will be the *baseline* case for all examples in this manuscript. It is an example of a Von Neumann architecture: memory holds both the *code* for task *T* (the instructions that implement its computation) and its *data*. Virtually all modern processors can be depicted as per Fig. 1; however, most processors include (at least one, potentially several) cache memorie(s) to accelerate execution. We'll ignore caches for the time being.

The notion of *performance* of task *T* is context-dependent: in some situations, *latency* (average or maximum time between an input arrival and output processing) is an adequate measure of performance; in others, *throughput* (number of inputs processed per time unit) is a better one. For simplicity, we'll assume task *T* must read *D* data (in bytes), process them, and write *D* data back to memory, and that the *total latency* (total execution time) is a valid measure of its performance.

Reading/writing data (including instructions) from/to memory requires a certain amount of time, determined by the specific implementation of the memory system and the specific access patterns. Processing that data (executing instructions), similarly takes a certain amount of time, determined by the specific implementation of the processor, and by the computation that must be performed. Determining the total execution time is not a trivial thing, as there are several sources of parallelism that make these inner latencies overlap (e.g., pipelined processors might be reading instructions from memory at the same time as they are processing previously read instructions; cache memories further complicate this timing analysis).

Task *T* is said to be *compute-bound* if making the memory system arbitrarily fast would have little to no influence on the execution time; the total latency is primarily a function of computation time (data processing). *T* is *memory-bound* if making the processor arbitrarily fast would have little to no influence on the execution time: the total latency is primarily a function of memory access times. Most real-world workloads are likely neither memory- nor compute-bound, meaning that making either the processor or the memory system faster would accelerate their execution; however, most workloads are probably far closer to being memory- than compute-bound (thus the heavy use of cache memories in real processors).

The performance bounds of a task have implications on its hardware acceleration: let's examine the two cases.

2.4.1 Accelerating Compute-Bound Tasks

If task *T* is compute-bound, we can replace its software implementation by an equivalent hardware pipeline: the computed *algorithm*, rather than being implemented as a set of instructions, becomes a circuit with equivalent behavior, as depicted in Fig. 1. The relevant question, of course, is "Is this implementation faster than the original one?". Unsurprisingly, the answer is: it depends.

If the implementation technology for the old processor and the new circuit is the same (e.g., both implemented in FPGA logic) the answer is probably yes. The circuit does not need to fetch and decode instructions: what would have taken several clock cycles in the software implementation becomes a single clock cycle in the hardware version. Despite processing the same amount of data, in exactly the same way, the logic that computes those data is much faster (requires fewer clock cycles), and the clock frequencies between the software and hardware versions are likely approximately the same.

If the implementation technology is different, things are less clear. For example, let's say we're converting a software implementation running on a processor clocked at 2GHz, and our resulting FPGA implementation is clocked at 500MHz (FPGA implementations always boast lower clock frequencies than the same circuit implemented on custom silicon). In this example, clock frequency goes down by a factor of 4. The hardware implementation will be faster if, and only if, the number of required clock cycles to compute the algorithm goes down by a factor greater than 4. *This is algorithm-dependent*: in fact, it's also instruction set dependent and micro-architecture-dependent. In summary, it is not trivial to determine whether a task will be accelerated when we move across implementation technologies.

Data parallelism Of course, there is no reason to implement an "equivalent hardware pipeline": it is a much better idea to implement an optimized hardware pipeline. One of the obvious optimizations is parallelism.

Consider the following code:

Listing 1: Data parallel task

```
1 int array[ARRAY_SIZE];
2 ...
3 for(i=0; i < ARRAY_SIZE; i++)
4 {
5    array[i] = foo(array[i]);
6 }
```

If task T computes the code depicted in Listing 1, it is completely useless, in a single-core software implementation, to parallelize the computation (i.e., through multi-threading); performance would actually decrease, as there would be no acceleration whatsoever (threads execute in temporal parallelism) and we would induce overhead in thread creation, etc. (in a multi-threaded/core processor scenario, any possible improvement would depend on the size of data, cache behavior, etc.: multi-threading is useful for hiding I/O latencies, not for data parallelism).

However, as we move the task to hardware, and still assuming it is and would remain compute-bound, then we could truly parallelize data processing: we could create a circuit consisting of N identical computational pipelines (circuits execute in spatial parallelism), accelerating computation by a factor of N (as long as the system remained compute-bound).

Instruction parallelism In single-core software implementations, the code depicted in Listing 1 would execute under instruction-level parallelism in most modern processors (pipelined or superscalar). I.e., the instruction stream responsible for said code, when executing on the processor's datapath, would temporally overlap, across pipeline stages or superscalar execution units (in- or out-of-order, depending on the architecture). This has been a standard performance acceleration technique for decades, extremely effective when there are few dependencies between instructions.

As we move the task to hardware, pipelining execution remains a standard technique, allowing hardware to execute several parts of the execution at the same time (most typically called temporal, rather than instruction-level, parallelism, since the concept of instructions no longer applies). Execution tends to be further optimized, since dynamic dependencies that must be resolved by processor hazard logic in processors, for software implementations, can be resolved at design-time for hardware accelerators.

Task parallelism This is the form of parallelism where the differences between software and hardware implementations are most clear. When comparing software solutions on single-core processors, task parallelism is merely instruction-level parallelism, at much coarser granularity. When transposed to hardware, independent tasks can be performed each using dedicated, truly independent hardware circuits. Thus, hardware acceleration is truly the transformation of temporal into spatial parallelism.

More commonly, now that multi-core processors are ubiquitous, software tasks are allocated to different cores. Thus, task-level parallelism, in the context of hardware acceleration, is nothing more than the union of the acceleration of each independent task (subjected to the aforementioned parallelization strategies).

2.4.2 Accelerating Memory-Bound Tasks

It is worth re-iterating that most real-world workloads are probably far closer to being memory- than compute-bound: the memory system, not the computing system, is almost always the performance bottleneck. In this scenario, all the advantages of hardware acceleration presented so far are negated: even spatial parallelism is useless, since the pipelines are starved of data, and must wait for memory. *Naively converting a memory-bound task to hardware yields no performance improvement whatsoever*.

Consider, as an extreme case, random sampling of points from an image, as would be performed on the Random Sample Consensus (RANSAC) algorithm [23]. Code might look something like:

Listing 2: Random sampling of points from image

```
1 int *input_frame;
2 int sample[N_POINTS];
3 ...
4 for(i=0; i < N_POINTS; i++)
5 {
6     int k = rand() % (WIDTH * HEIGHT);
7     sample[i] = *(input_frame + k);
8 }
```

This is a clear memory-bound task. Little computation is performed (assuming random index generation is negligible): the bulk of the time is spent in accessing external memory for point sampling. Because of its random nature, access is likely to exhibit poor cache behavior, since there is little to no temporal or spatial locality, assuming image size is significantly larger than cache line size. Thus, any attempts at acceleration through hardware will result in no performance improvement whatsoever, since the bottleneck (memory access) remains unaltered.

The advantages of hardware acceleration come, not from naive conversion from software to hardware, but from using

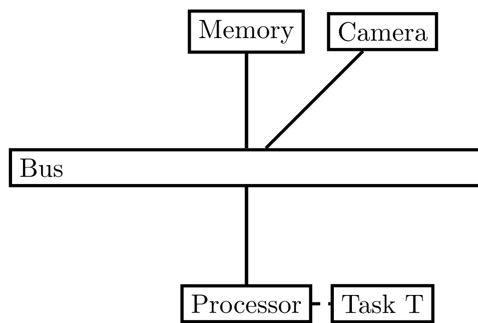

Figure 2 Camera connected to system bus. Bus data transfer quickly becomes the performance bottleneck, as it does not take advantage of any specificities.

this newly-available hardware to re-design how our system is architected (i.e., *refactoring*). One of the most powerful refactoring approaches comes from realizing that **no data are born in memory**: data are born from system inputs. If we have custom hardware available, we can propagate and store those data in different ways than just a monolithic memory, resulting in far more efficient implementations.

Consider the system depicted in Fig. 2: a camera is connected to the system bus, providing real-time video streaming. A configuration such as this typically employs Direct Memory Access (DMA) to transfer image frames from the camera directly to memory, without the need for processor polling; camera writes data in row-wise order. Let's assume a simple region-wise image processing application, e.g., edge detection. Task T merely sees frames stored in memory (as a 2-dimensional array) that can be read and processed, generating output that is written back to a different memory region. Software implementation looks like this:

Listing 3: Edge detection: software implementation (edge cases ignored)

```
int *input_frame;
int *output_frame;
...
#define F input_frame
for(i=1; i < HEIGHT-1; i++)
{
    for(j=1; j < WIDTH-1; j++)
    {
        int Gx = F[j-1][i-1] + 2*F[j-1][i] +
            F[j-1][i+1] ... - F[j+1][i+1];
        int Gy = ...
        output_frame[j][i] = sqrt(Gx*Gx +
            Gy*Gy);
    }
}
```

This is an example of Sobel edge detection, applies a 3x3 sliding kernel to the input image. The specific computation is not important for our purposes: what is important is to observe data access patterns. For an input frame of width W and height H, this task would generate an output frame of width $W-2$ and height $H-2$ (frame edges are ignored). The double *for* loops traverse the output frame row-wise. For every j, i coordinate pair in the output frame, we must read the 8 adjacent values from the input frame, from $j-1, i-1$ to $j+1, i+1$.

A naive conversion of this task to hardware, connected to memory in the same way, would result in a virtually identical implementation: the system would have to perform 8 memory reads for each $(W-2)*(H-2)$ memory writes, after the camera had already written $W*H$ data. In total (ignoring instructions) the system would have to access memory $W*H$ (camera) $+9*((W-2)*(H-2))$ (processing) $\approx 10*W*H$ times.

Custom memory-processing systems A more sophisticated hardware acceleration would take advantage of two observations: (1) the input frame is discarded after edge detection is computed. Thus, if we can process data in a streaming manner (as it is read from the camera), there is no need to write the input frame to memory. (2) the input frame data access pattern is regular: thus, it must be possible to implement a data storage system that allows us to process it more efficiently. This solution is depicted in Fig. 3.

Figure 3 depicts a shift-register *row-buffer*. Notice that it uses the smallest amount of storage required to compute the Sobel edge detection operation, taking advantage of distributed memory inside FPGAs to accelerate the computation, and discarding pixels as soon as they are no longer required. In this configuration, camera still has to write to FPGA memory (total of $W*H$ writes), but configurable logic needs only write the resulting frame $(W-2)*(H-2)$ to memory, for a total of $\approx 2*W*H$ memory (main and FPGA internal) accesses, resulting in a 5x improvement over the original implementation.

3 High Level Synthesis

High-Level Synthesis (HLS) promises to boost hardware design productivity into software-like levels, ending the long development and verification processes typically associated with Register Transfer Level (RTL) design. The momentum behind the HLS movement is primarily fueled by two aspects: on one hand, the hardware community desires to increase productivity and ensure design correctness in order to guarantee the time-to-market of complex SoCs keeps up with Moore's law. On the other hand, the software community, lacking RTL-savvy, increasingly adopts FPGAs in fields as diverse as high performance computing or image processing, and

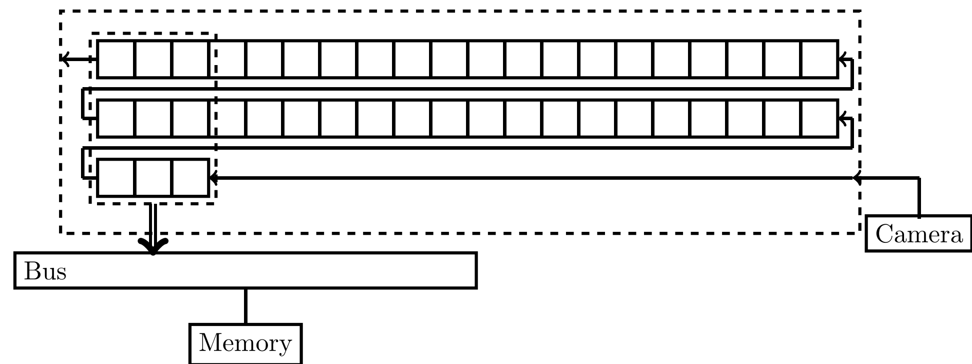

Figure 3 Camera connected to FPGA with custom memory system (width of shift register is *W*).

wishes to program them using established software languages/frameworks. The appearance of *platform FPGAs*, which are essentially SoCs combining general-purpose processor(s) and/or other "hard" blocks with "soft" FPGA fabric, increased the demand for powerful HLS even further, thanks to opportunities for hardware-software co-design.

Despite approximately three decades of HLS development, we are still far from the levels of productivity achieved in the software world. The HLS landscape, shaped by FPGA vendors' proprietary toolchains, third-party commercial toolchains and academic/open-source projects, is made up of more than forty different HLS frameworks, none of which has gained widespread adoption: Verilog and VHDL are still the *de facto* development languages for FPGAs. The goal of this survey is to examine the reasons behind this and point out future research directions for HLS.

HLS research has been previously summarized from different perspectives: [24–27] and [28] have described the historical evolution of HLS tools, primarily focusing on industry adoption. Zhang and Ng [29], Compton and Hauck [30] and Cardoso et al. [31] focus on the dynamic-reconfiguration support of HLS tools. We summarize the field from a different perspective, filling a gap in the literature; namely *HLS languages' abstractions*, focusing on the clash of hardware and software traditional views.

The diverse nature of HLS languages/toolchains (some are novel full-fledged languages such as Bluespec, others subsets of legacy languages such as C, and others are domain specific embedded languages, such as Cλash) complicates this analysis; we treat each as comparable languages *per se* when appropriate, and we make very fine distinctions in other cases. Different toolchains which use the same underlying language are also treated distinctly: throughout this manuscript, we use the terms *language* and *toolchain* interchangeably when referring to HLS.

Hardware Description Languages (HDL), such as Verilog and VHDL, lack many of the features typically associated with high level languages. There is no complex type system: every signal is a bit array (integer types are merely placeholders for a specific bit width). There are very few syntactic constructs: the various loop and type constructs typically found in software make no sense. This is because HDLs specify the behavior, not as a sequential computation (where parallelism must be explicitly managed through threads/processes) but as a computation for every single moment in time (behavior at each clock cycle) where parallelism is implicit. Loop unrolling is seen as a compiler optimization in C, while in HDL, it is built into the language semantics: a loop that cannot be fully unrolled at design time will result in a synthesis error. This paradigm shift from temporal flexibility to temporal strictness is one of the main reasons software programmers struggle with HDLs, and one HLS tools attempt to abstract.

However, this temporal paradigm is one of the greatest strengths of HDLs and one of the weaknesses of HLS (we will elaborate on this in the following sections); HDLs can cope with complex timing requirements, such as refresh rates for DRAM memories, communication protocols, or circuit initialization procedures. When HLS languages fail to provide mechanisms to perform these procedures efficiently and clean interfaces to HDL code, there is little motivation for hardware engineers to adopt such a toolchain. This advents from the fact that many HLS languages are datapath-oriented, rather than control-oriented (HDLs are both).

Clock signals are arguably the most important signals in an FPGA design, and thus, they are explicitly stated in HDLs. Designers can distribute complex designs across asynchronous clock domains, perform clock gating for power reduction and keep an accurate record of elapsed time (every hardware designer has had to implement a clock cycles counter for some low-level communication protocol at one time). Most HLS languages, however, do not have an explicit clock signal. Either each language construct operates in one clock cycle or, as is the case in some dataflow languages, there is no programmer-visible concept of time. To the best of our knowledge, very few HLS toolchains are yet capable of handling multiple clock domains automatically, inserting appropriate synchronization logic (e.g., [32]).

3.1 Imperative Languages

C is the most familiar software language for hardware designers. Hence, it is not surprising that many HLS toolchains use C, and other similar imperative languages and Object-Oriented (OO) variants, as a foundation (although many software programmers would disagree C should be considered "high level"). Table 1 depicts imperative-based HLS toolchains. It is by no means complete, but suffices to grasp the amount of effort invested in imperative languages to FPGA design.

There are three main advantages to this flavor of HLS: *familiarity, control* and *co-design*. The familiar syntax allows programmers to express algorithms quickly, often re-using code. Tried and tested software applications can be migrated to hardware for acceleration. The sequential semantics of C-like languages, rich in loop and conditional constructs, lends itself well to the implementation of protocols and control operations. Since software and hardware are described in the same language, design space exploration strategies can be employed to meet design performance, power and area constraints, mapping an algorithm to CPU(s)/FPGA hybrid systems. This paradigm is especially favorable when targeting platform FPGAs; hence the extensive support offered by FPGA vendors (e.g., Xilinx Vivado HLS and SDK).

Table 1 Imperative/OO HLS.

Language	Source
SystemC	[33]
Handel-C	[34]
OCAPI-XL	[35]
Catapult-C	[36]
Vivado HLS	[37]
Impulse C	[38]
C-to-Silicon	[39]
Synphony C	[40]
Cynthesizer	[41]
LegUp	[42]
ASC	[43]
Altera C2H	[44]
CHiMPS	[45]
ROCCC	[46]
GAUT	[47]
Trident	[48]
Altera SDK for OpenCL	[49]
Xilinx SDAccel	[50]
FCUDA	[51]
LIME	[52]
KIWI	[53]
DWARV	[54]
Bambu	[55]
Hercules	[56]

This paradigm is not without several drawbacks. It is notoriously difficult to explicitly express parallelism in imperative languages; it must typically be expressed through compiler directives (*pragmas*) for loop unrolling, which rely on compiler optimizations. Granularity is an issue: most HLS compilations operate at the function granularity, which might force legacy software to be re-written in order to encapsulate hardware-destined and software-destined code separately. Timing is an issue, as it may be impossible to predict how many clock cycles a particular language construct will take to execute, depending on the HDL generation strategy. As these languages were built for Von Neumann machines (implying a large, shared memory space), many language features are not directly amenable to hardware synthesis (pointers and dynamic memory allocation are notorious examples).

3.2 Functional Languages

Advocates for functional programming have developed several HLS functional flavors, either through new complete languages or through small languages embedded in consolidated ones such as Haskell. Table 2 depicts an overview of functional languages targeting FPGAs.

In some ways, functional languages are a better fit for HLS than imperative languages. The referential transparency that makes functional languages inherently suitable for parallelism lends itself well to hardware synthesis. Functional HDLs like CλaSH are able to model functions as structural definitions of circuits and function applications as instances of those circuits. The fundamental support for higher-order functions means that designs are often automatically parametric. In many ways, functional languages constructs can be mapped almost directly to RTL, allowing a designer working in a functional HLS language to have a very good idea of exactly what the hardware they are writing will look like, in contrast to imperative languages where there can be a much larger gap between the code and the hardware generated by synthesis tools.

There are also a number of disadvantages to using functional languages for high level synthesis. Some constructs used in functional programming are not easily synthesizable. E.g., recursive functions can pose particular difficulties; if function applications are realized by instantiating hardware components, a recursive function would conceptually result in an infinitely large hardware construct. It is possible to identify certain special cases of recursion and generate specific hardware for them, but it is difficult to find a general solution to the problem. Similarly, it is generally not possible to realize recursive data types in hardware. These limitations can be an issue for programmers who are used

Table 2 Functional HLS.

Language	
CλaSH	[57]
HML	[58]
ForSyDe	[59]
Lava	[60]
PARO	[61]
Esterel	[62]
MMAlpha	[63]
Verity	[64]
ReWire	[65]
SAFL	[66]
Hume	[67]

to functional languages. As they may need to learn a very different approach to many problems then they are used to when writing software, this can negate some of the advantages of high level synthesis.

3.3 Domain Specific Languages

In an effort to avoid the challenges of synthesizing complete complex languages, several Domain Specific Languages (DSL) have emerged in recent years. DSLs are well known in the software domain: they allow programmers to express solutions at a higher abstraction level than typical general-purpose languages, within the semantics of the particular application domain.

In the context of HLS, DSLs have several other advantages. The limited syntactical constructs are more easily synthesizable; in other words, it is far simpler to ensure complete language coverage. Knowledge of the application domain allows the use of hardware templates optimized for the domain: rather than generic hardware structures designed for flexibility, the HLS tool is free to infer specialized architectures, optimized for power, performance and area within the domain (e.g., in synchronous dataflow DSL synthesis, the HLS compiler is aware of the timing relationships between modules and can infer pipeline stages separated by registers; in asynchronous dataflow, the HLS compiler is forced to infer FIFOs between modules).

Darkroom [68] is language and compiler for image processing. Its semantics allow it to synthesize line-buffered pipelines, with all intermediate values in local line-buffer storage. Images at each stage of computation are specified as pure functions from 2D coordinates to the values at those coordinates, declared using a lambda-like syntax. In the first version, it only supports programs that are straight pipelines with one input, one output, and a single consumer of each intermediate value. HIPA^{cc} [69] (Heterogeneous Image Processing Acceleration) is another DSL for image processing. It is a C++ embedded DSL [70], which uses the LLVM back-end [71] for software code generation, and C code annotated with pragmas for Vivado HLS.

RVC-CAL [72] is an asynchronous dataflow language which possesses backends for FPGA, e.g., Xronos [73]. RVC-CAL is based on dataflow process networks with the addition of firing rules. Xronos uses Orcc compiler [74] as its front-end, which parses RVC-CAL actors and generates an intermediate representation suited to its OpenForge back-end. Another streaming DSL is Optimus [75], based on the StreamIt language [76]. A common feature of both languages is that embedded memories are used to implement local arrays and other data structures used by the filters. Thus, for large stream graphs, embedded memories quickly become the bottleneck resource.

Spiral [77] is a code generation system for Digital signal Processing (DSP) transforms which has been extended to generate DSP IP cores for FPGA [78]. RIPL [79] employs image processing algorithmic skeletons as a general framework for the application of user-defined functions, generating efficient streaming hardware pipelines.

3.4 New Generation Hardware Description Languages

Three new generation HDLs are particularly relevant: Bluespec [80], Chisel [81] and Cx [82].

Bluespec extends SystemVerilog to provide a higher level of abstraction. Interfaces are a core construct of Bluespec. Interfaces group signals according to methods, which define the semantics of access to a signal and can be used to parameterize modules upon instantiation. Rather than ”always" or ”process" constructs familiar to Verilog/VHDL designers, Bluespec defines behavior through rules (i.e., guarded actions) which specify how data are moved from state to state.

Chisel is an HDL embedded in the Scala language which supports multiple design paradigms, including object orientation, functional programming, parameterized types, and type inference. Two notable features are the capability to specify composite types (i.e., C-like *structs*) and automatic inference of bit-widths, unlike strict bit-width definitions in legacy HDLs. OO-like inheritance allows modules to be re-used in the definition of higher-order modules without the messy sub-module instantiation (and corresponding ”rats nest" wiring). Computations can be expressed in functional-friendly constructs such as *map* and *fold* and the expressive generator systems simplifies the re-use of parameterizable modules.

Cx offers a highly structured syntax with strong bit-accurate static typing. A Cx design is described as a set of sequential tasks connected together and executed concurrently, where dependency injection and inheritance can be applied to tasks. The Cx compiler generates human readable Verilog/VHDL code or C code for verification. Cx systems are described as Kahn Process Networks [83] where

connections are inferred by discrete non-interruptible execution rules. Unlike legacy HDLs, clocks are not explicitly declared within code, but language semantics strictly specify the timing behavior of language constructs, providing cycle aware design much like VHDL and Verilog.

Bluespec, Chisel and Cx are greatly superior to legacy HDLs. Software concepts such as inheritance and type composition have found the way to HDLs, reducing the semantic gap between concept and implementation. Rather than leveraging existing software languages for FPGA synthesis (with the associated semantic problems), new generations HDL incorporated high-level language concepts in languages fine-tuned for hardware design. However, these new generation HDLs are still far from producing the type of disruptive innovations expected from HLs, primarily due to three reasons:

- They follow the same design principle as decades-old legacy HDLs: FPGAs as a self contained entity. This contrasts the approach offered by FPGA vendors, who provide software suites that allow design at board, rather than chip, level. These suites are made up of several stacks which, through a complex design process involving constraint files, IP libraries, proprietary buses, etc, generate FPGA designs incorporating peripheral devices access and software interaction. It would be expected of new generation HDLs to incorporate more complex constructs, modeling off-chip interfaces within the semantics.
- They do not incorporate the Von Neumann notion of memory. A substantial portion of state of the art FPGA systems incorporate external memory to accommodate data requirements. On new generation HDLs, this must be handled as in legacy HDLs: the programmer is responsible for interfacing with memory and managing data transmission to and from, burdening them with implementation details independent of the top level computation. It would be expected of new generation HDLs to model memory transparently (e.g., by specifying memory-allocated data as a built-in type) and generate hardware to transfer to and from memory seamlessly, through an on-chip hierarchy (physical constraints such as which FPGA pins are connected to memory can be resolved at linking stage, prior to synthesis).
- There are no semantic considerations for software interface. With the rise of the platform FPGA, it would be expected that new generation HDLs would borrow concepts from research in hardware-software interface research (e.g., PushPush [84]) in order to provide semantic mechanisms for incorporating typed functions for bi-directional interaction with software objects.

3.5 HLS Evaluation

To compare the implications of different HLS paradigms, we implement two computations that are ubiquitous in the image processing domain: Finite-Impulse Response (FIR) filter and Eigenvalue/vector decomposition.

Since the HLS-acceleration of complete algorithms is significantly influenced by algorithm particulars (including, e.g., input size), we instead focus on evaluating partial computations that are applicable to several classes of representative solutions. E.g., FIR filters are applied within large groups of smoothing and sharpening applications, whilst Eigenvalue computation plays prominent roles in algorithms such as Principal Component Analysis and Optical Flow. Thus, their analysis provides valuable insights to designers interested in accelerating their own algorithms.

An FIR filter in discrete-time domain is defined in Eq. 1, where $y[n]$ is the output, $x[n]$ is the input sample, $h[n]$ is the coefficient and L is the number of filter taps.

$$y[n] = x[n] * h[n] = \sum_{i=0}^{L-1} x[i]h[n-i] \quad (1)$$

Eigenvalue/vector decomposition can be implemented through the Approximate Jacobi method, replicated from [85]. If $A\mathbf{x} = \mathbf{b}$, where A is an $n \times n$ matrix, and x and b are column-vectors of length n, then A can be decomposed into a diagonal component D, a lower triangular part L and an upper triangular part U such that $A = D + L + U$. The solution can be obtained iteratively via $\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - (L + U)\mathbf{x}^{(k)})$.

Table 3 presents results for FIR filter implementation through CλaSH (Haskell), Vivado HLS (C and SystemC), without any optimizations performed; Table 4 lists the same performance metrics obtained in the second scenario where we used optimization directives for parallelization. Tables 5 and 6 present the same results for Eigenvalue/vector decomposition.

Table 3 Standard Optimization Result (FIR Filter).

	Haskell	C	SystemC
BRAM	0	0	0
DSP48E	0	2	2
FF	640	112	119
LUT	1235	129	200
Static Power (W)	0.574	0.123	0.123
Dynamic Power (W)	23.370	0.002	0.003
Latency(cycles)	10	617	248
Latency(absolute) (ns)	100	6170	2480

Table 4 Performance-Optimized Results (FIR Filter).

	Haskell	C	SystemC
BRAM	0	0	0
DSP48E	0	0	0
FF	640	522	633
LUT	1235	686	894
Static Power (W)	0.574	0.123	0.123
Dynamic Power (W)	23.370	0.013	0.004
Latency(cycles)	10	14	27
Latency(absolute) (ns)	100	140	270

Table 5 Standard Optimization Result (EVD).

	Haskell	C	SystemC
BRAM	0	4	2
DSP48E	12	11	38
FF	388	2185	5795
LUT	7519	3122	8624
Static Power (W)	0.157	0.123	NA
Dynamic Power (W)	4.999	0.017	NA
Latency(cycles)	10	1860	8130
Latency(absolute) (ns)	100	18600	81300

Results are noteworthy, and particularly relevant to designers who want to select a HLS approach for their implementations. CλaSH regularly provides the best performance (latency), at the expense of size and power consumption (as a function of its greedy parallelization approach). In contrast, C and SystemC based approaches give more balanced results; their strategy is to minimize chip area by reusing components for computation, at the expense of longer processing time. These tradeoffs should guide adoption of HLS framework.

4 The IMP Language Architecture

IMP-lang (the IMage Processor Language) is a design space exploration and simulation language, useful for early design partitioning across heterogeneous cores. Its features are particularly relevant to the design of streaming systems, such as typically found in the embedded image processing domain, but it can be applied to other domains. The language compiler and interpreter is open-source, and can be obtained here.[1]

IMP-lang code is compiled to an intermediate representation (IR), where each IR sequence comprises the code required for each parallel computation ("task") in the IMP-lang abstract machine, as well as constructs required for communication and synchronization of each task. The interpreter executes IR code, deciding when to execute each task in function of code semantics and runtime operation, managing communication and synchronization (e.g.,, communication between tasks at the highest abstraction level corresponds to calls to the interpretation engine for data transfer and notification). The interpreter aids designers in performing early design space exploration of their design options across heterogeneous deployments, through iterative compilation/evaluation loops. A finalized design can be committed to hardware and software by replacing the interpreter with appropriate back-ends for hardware and software code generation.

[1] https://github.com/paulofrgarcia-cmkl/IMP

4.1 Cores

A "core" is the primary unit in IMP-lang. It abstractly represents a piece of hardware that can execute code. For example, a CPU, a GPU unit, or a bespoke accelerator, could all be represented as a core. Each core has code assigned for it in the form of functions (see Section 4.2). Each core should execute independently of the others, as they represent separate hardware units. A core is considered to be active as long as at least one of its functions is running; if no cores have running functions, the program is finished (see Fig. 4).

4.2 Functions

There are three categories of functions: Main, Signal, and Stream functions. These will be elaborated on in the sections below.

4.2.1 Main Functions

The main function is the primary function of each core, and every core requires one and only one main function. It runs by default, and once the core's main function reaches the end of its code, or the function is terminated, the entire core is considered to have finished its work and stops all

Table 6 Performance-optimized Result (EVD).

	Haskell	C	SystemC
BRAM	0	0	4
DSP48E	12	28	41
FF	388	24455	6179
LUT	7519	28020	13876
Static Power (W)	0.157	0.124	NA
Dynamic Power (W)	4.999	0.170	NA
Latency(cycles)	10	135	824
Latency(absolute) (ns)	100	1350	8240

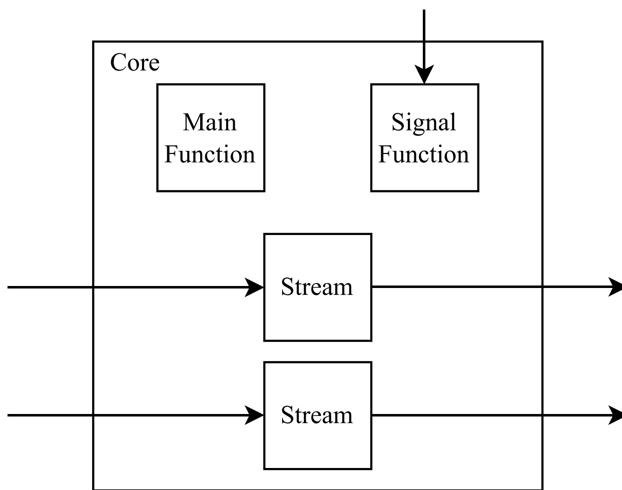

Figure 4 Illustration of a core with a main, signal, and two stream functions.

execution. As a result, if the core needs to remain idle until some condition occurs (for example, a core representing a co-processor that does nothing until triggered, computes a value, and then returns to an idle state), the main function should wait in a while-loop lasting until the end of the simulation (see Fig. 5).

4.2.2 Signal Functions

Signal functions are analogous to interrupt service handlers. If a core is given a signal function, and its main function enables signal interrupts, the core will run its signal function in response. It should run exclusively once triggered; that is, other functions must wait for it to finish before they can continue; Figs. 6 and 8 illustrate this blocking effect. A signal function can alter its behavior depending on which core triggered it, allowing for a single core to respond appropriately to many different circumstances and hardware layouts.

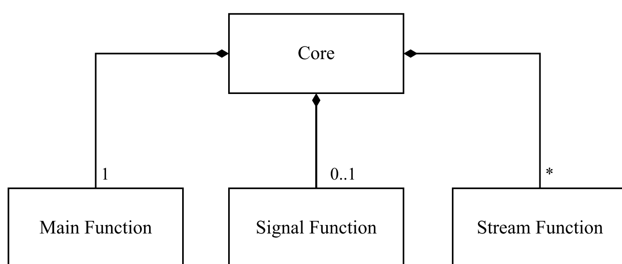

Figure 5 Class diagram illustrating the relationships between cores and the three function types.

4.2.3 Stream Functions

Stream functions serve the role of asynchronous subroutines. They are composed of a block of executable code, an input queue, into which arguments are passed, and an output queue, onto which return values of the function can be pushed. The stream function can be invoked by any other function, including another stream function, or even itself. By invoking it, the arguments supplied to the call are collected and pushed onto the input queue, the capacity of which is infinite (by default in simulation, but can be modified in implementation to better suit the realities of target hardware), so that the function can be invoked at any time. Any argument passed to or returned from a stream function can be null-valued; function code can be written to behave differently when encountering a null argument, and so null arguments can be used to convey information (such as indicating that an error occurred, that a search returned no results, or as a flag passed to a stream function to indicate that it should perform an alternative set of operations on other input arguments).

Once input data is at the head of the input queue, the function's code will execute, in parallel to the other functions on the core (though it will still be paused while a core is handling a signal interrupt). Once it reaches the end of its code, it will terminate, and will push values of any outputs that it is defined to return onto its output queue. Just as any function of any core can write to the input of a stream function, the data at the front of the queue is visible publicly. Thus, any function can read from the queue; this destructively consumes the value at the front. The architecture of a stream function and its input/output queues is illustrated in Fig. 7.

A function may attempt to read from an output queue at any time. However, if the queue is empty, that function will block until data is present in the queue. This access is on an opportunistic basis; if multiple functions are attempting to access a queue, the next one to execute will take its turn first. This is not an issue if all functions are only reading, as the read process is non-destructive. If one or more of the competing functions is waiting to discard a value from the queue, however, this can result in destroying data that other functions are expecting to read, with the other functions then erroneously reading the next value in the queue, or blocking if the queue has been left empty. These sorts of conflicts are presently the responsibility of the developer to avoid; implementation-specific means to these conflicts are possible, but currently unimplemented (see Fig. 8).

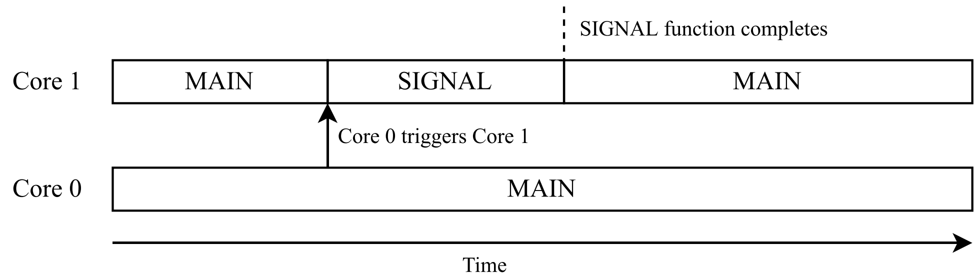

Figure 6 Timing diagram illustrating the program flow on a core when receiving a signal interrupt.

4.3 Inter-core Communication

IMP-lang provides two methods for cores to intercommunicate: stream and signal functions. Signal functions can be used to emulate interrupts from hardware peripherals such as keyboard, mice or other buttons, while stream functions can be used to emulate data transfers over a bus: for example, a primary core providing data to a core acting as a co-processor and awaiting the result.

A straightforward example of a system that can be built in IMP-lang is illustrated in Fig. 9. This system consists of three cores, and highlights a basic use of both modes of inter-core communication. The first is a main core, which could represent the main processor of a computer. The second is a timer core, which counts upwards until hitting a defined limit, signaling the main core upon finishing its count, then beginning a new cycle; this core could represent one of the timing modules built-in to the main processor. The third is a printer core, to which the main core sends an integer representing the current time to upon receiving a signal from the timer core, which it then prints. This could represent a hardware printer connected over a serial port, or the standard output stream of the operating system.

4.4 Hardware Synthesis

The constructs that make up IMP-lang are, by design, all directly amenable to straightforward hardware implementation. Cores, the primary unit of parallel computation in the implied model of computation, can be synthesized to independent hardware modules, corresponding to spatial parallelism. Internal functions represent further sub-divisions of parallel computation. Signals control hardware schedulers that enable or pause each computational pipeline, and trigger sub-computations, i.e., interrupts. Inter-core communication is achieved through abstract FIFOs at the language level, mapped to concrete hardware FIFOs once average occupancy has been determined through high level experimentation. Interfaces across hardware and software (consistently, at the language level, achieved though signals and FIFOs) can be translated to equivalent software/hardware interfaces in concrete

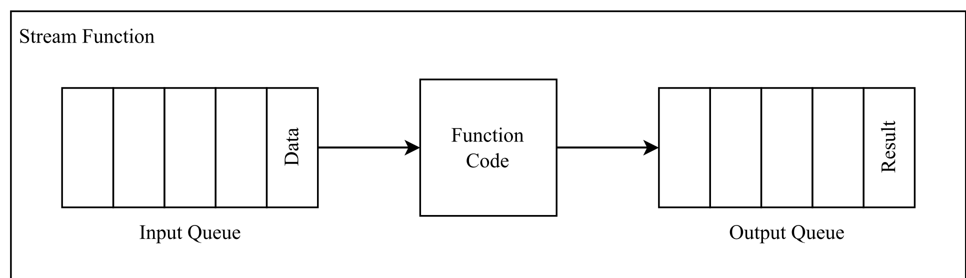

Figure 7 Architectural diagram of a stream function, illustrating its input and output queues.

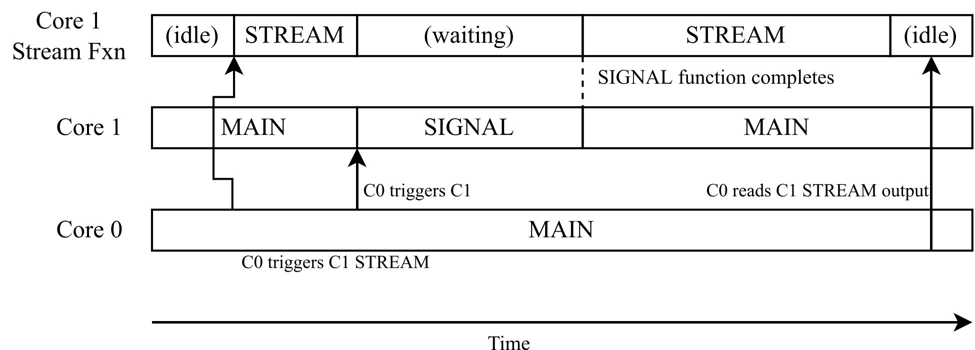

Figure 8 Timing diagram illustrating the program flow on a core and its stream function when receiving a signal interrupt. Note how Core 1's stream function also halts while the signal function is being handled.

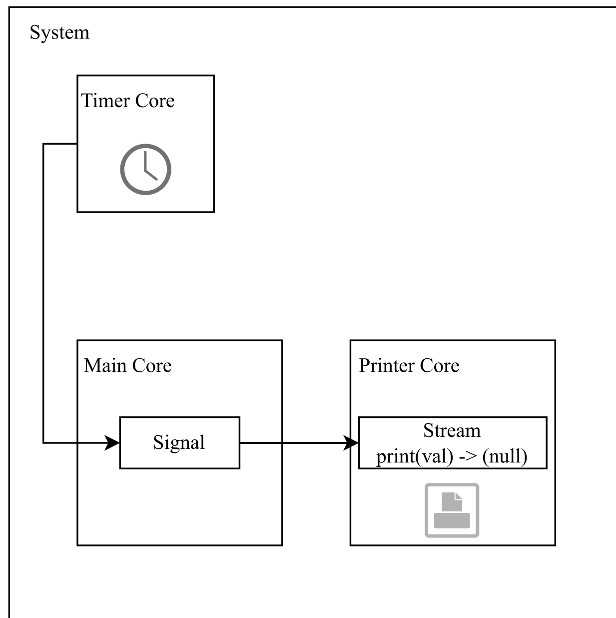

Figure 9 Block diagram of a sample system illustrating inter-core communication.

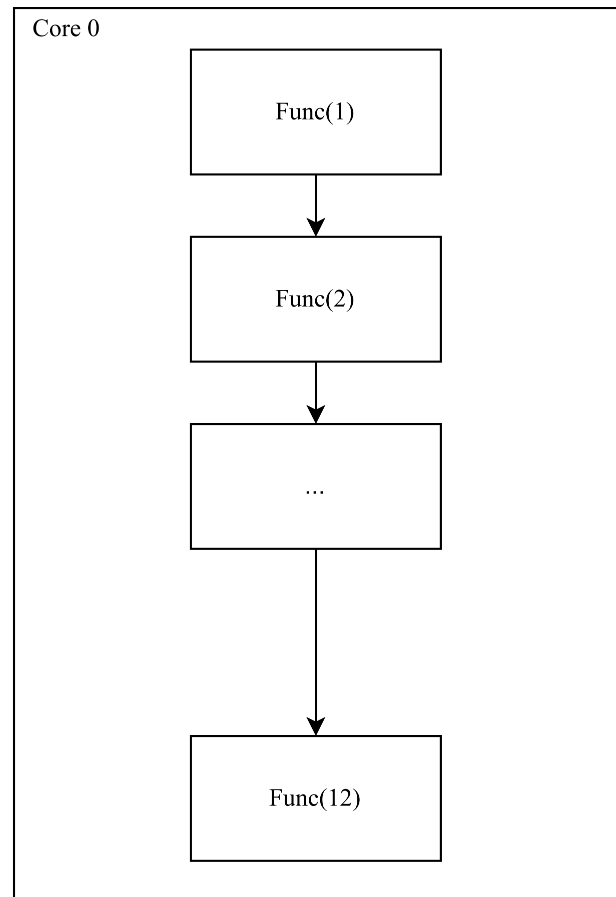

Figure 10 Program flow diagram of sequential function computation.

implementations. Although these are target system dependent, state of the art FPGA integration methodologies (e.g., Xilinx Xillybus [86]) use equivalent constructs.

5 Demonstrative Experimental Evaluation

5.1 Experiment 1

Let us imagine an arbitrary function, *Func*. We will use this function, run with inputs of increasing value, as an example to implement and compare three different strategies for coordinating parallel execution among a fixed number of cores. This could represent any computation that generates a new data frame from a previous one, where a data frame can be an image: for example, pixel-wise or windowing operations such as 2-D filters or a convolutional layer.

The first is the sequential approach, as illustrated in Fig. 10. It runs on a single execution unit, which computes and prints the output of each iteration of the function in turn.

The next approach introduces parallelism. It runs on three cores; two are identical, programmed with a stream function that executes the function, and the third acts as a coordinator. The coordinator core issues commands to the two computing cores to each compute *Func*, then collects the results and prints them. Since the coordinator core issues a command and then waits for the results, we call this a "synchronous" parallelism strategy (see Fig. 11).

The third approach takes advantage of stream functions' input and output queues. In it, instead of issuing and waiting for one command at a time, the coordinator core issues a batch of several commands at once to the computing cores and then collects and prints the results of the batch. This "batch-parallel" approach is illustrated in Fig. 12. This approach simulates asynchronous behavior across main-secondary computing units. Main unit issues as many requests (bundled with input data) as requires, regardless of secondary units' state. Requests sit in connection FIFOs, until such time when secondary units are ready to accept them. This illustrates the ease of separating concerns using FIFO interfaces (akin to dataflow HLS, as described in Sect. 3).

Each of these three approaches were run, and data was collected measuring the total execution time taken for computing *Func* for each input value, as well as the amount of time that the coordinator core (if present) spent blocked while waiting for the computation cores to finish their work. The results are detailed in Table 7.

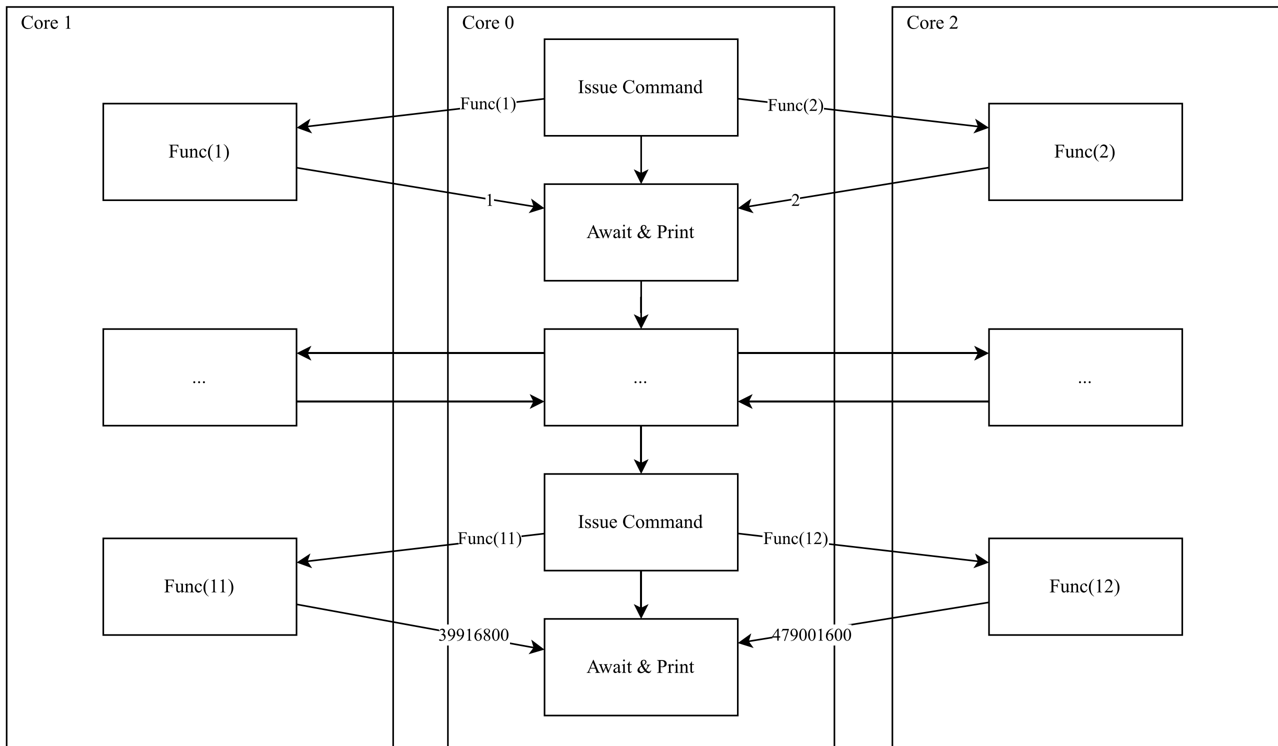

Figure 11 Program flow diagram of synchronous-parallel function computation.

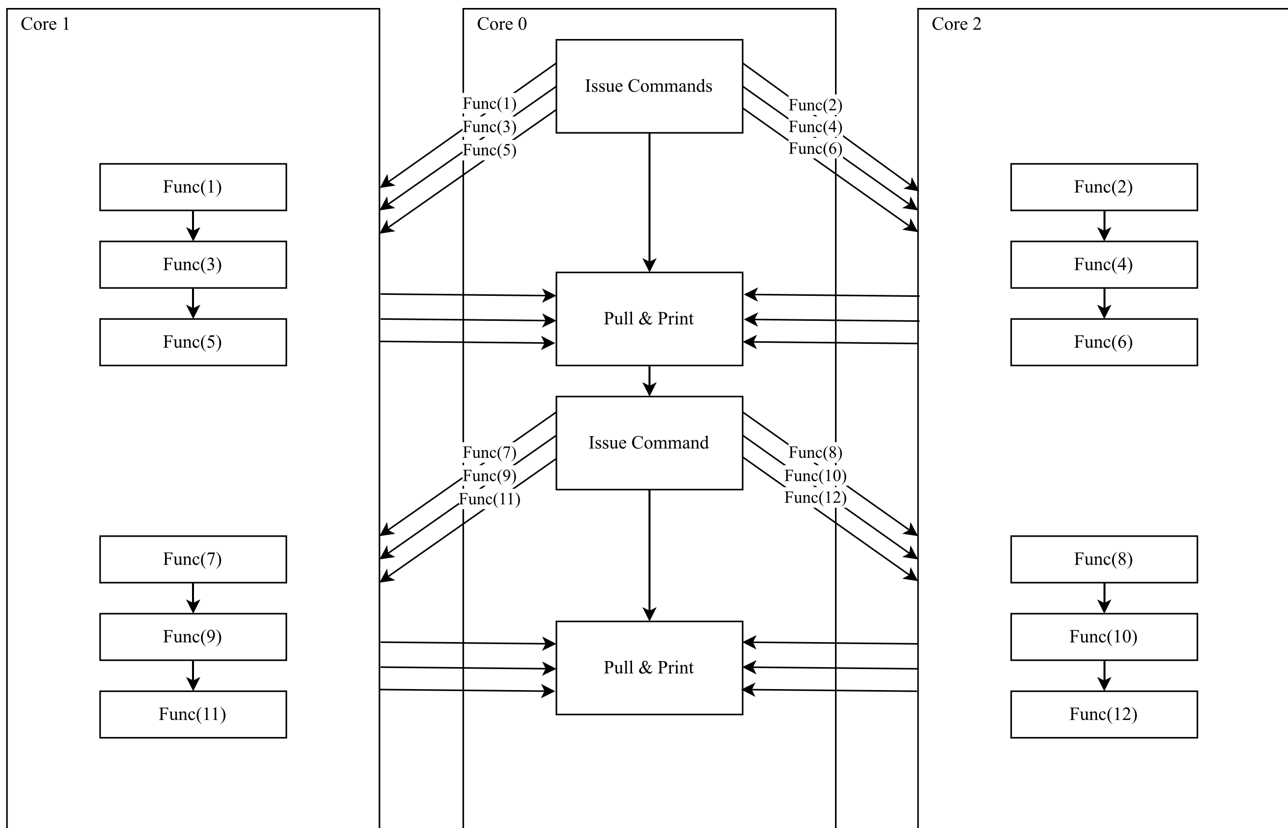

Figure 12 Program flow of batch-parallel function computation.

Table 7 Numeric Results of Experiment 1.

Strategy	Execution Time (Cycles)	Time I/O Blocked (Cycles)
Sequential	1279	N/A
Synchronous	738	576
Batch	654	520

5.2 Experiment 2

Here, we explore the performance impact of offloading computation onto multiple cores. We begin with a single core tasked with running three arbitrary functions, all of which take a single integer as input and return a single integer as output; these will be labeled F1, F2, and F3. We will then begin moving these functions to additional cores, and examining the effect on execution time and parallelism as offset by the increased cost in "size" incurred by adding more compute units.

The first design is illustrated in Fig. 13. The single core repeats a cycle of computing each function in turn, for inputs ranging from 1 to 49, printing the result of each. The execution times of the functions were profiled, revealing that F1 took at worst 31 cycles to run, F2 took at worst 131 cycles, and F3 took up to 649 cycles to complete.

The second design, illustrated in Fig. 14, adds a second core, onto which the function shown to consume the most time (F3) is offloaded by implementing it as a stream function that the original core calls, before computing F1 and F2; the intent is to allow the added core to compute the most time-consuming function in parallel.

The third design, illustrated in Fig. 15, furthers this strategy by offloading the second most time-intensive function (F2) onto yet another core. These parallelized designs are also run, executing each function in turn with inputs ranging from 1 to 49; data is collected on total execution time, the number of cycles in which execution is occurring in parallel, and the number of cycles that the coordinator core spends blocked. A sibling configuration was also run, wherein F1 is offloaded instead of F2. Results of this experiment are detailed in Table 8.

5.3 Results

In the IMP-lang interpreter, each round-robin cycle of core execution takes up one abstract time unit (whether that unit is a microsecond, second, or hour is irrelevant at this level of abstraction). Thus, the time a program takes to execute can be measured in terms of these cycles. As the simulation is fully deterministic, there is no issue of variance between runs to account for.

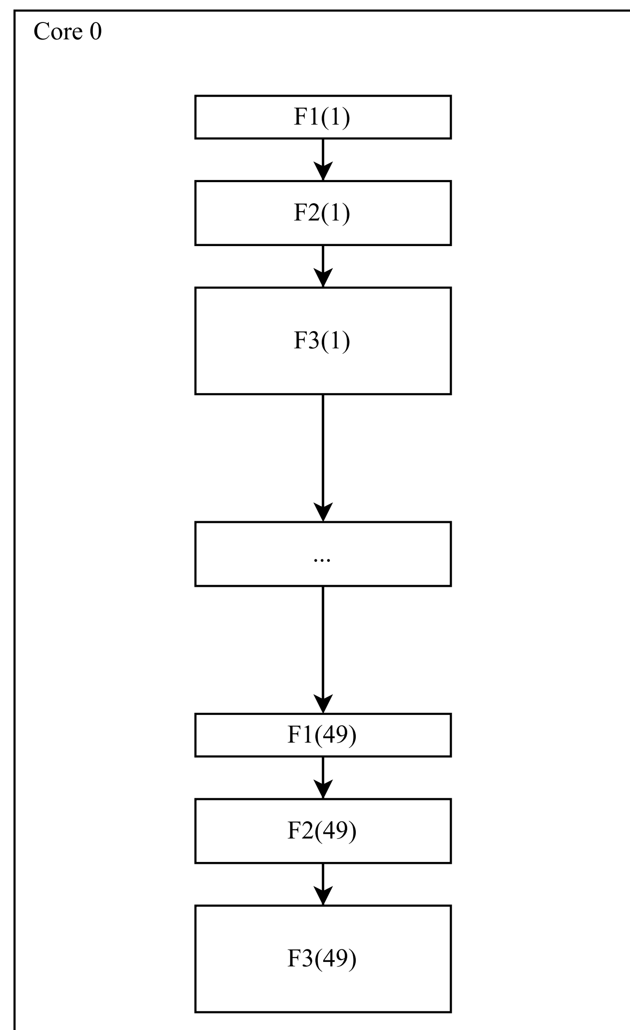

Figure 13 Program flow diagram of F1, F2, and F3 running on a single core.

Figure 16 displays the execution profiles of each of the core coordination strategies. The Sequential strategy took the longest to execute at 1279 cycles. This was expected, as it makes no use of parallelism; this lack of parallelism also means that the Sequential strategy cannot have spent any time I/O blocked. The strategies that do make use of parallelism (Synchronous and Batch) show significant improvements in execution, with Synchronous showing a 42% reduction in execution time, and Batch improving even further than Synchronous, with a 48% reduction in execution time. Batch also spent 9.7% less time waiting for its co-processors to finish their computations.

Overall, the Batch coordination strategy proved to be the most time-efficient of the three proposed. Crucially, this gain in performance was not difficult to achieve, as both altering the architecture of the system (creating the two computing cores) and tweaking the system's behavior (modifying the

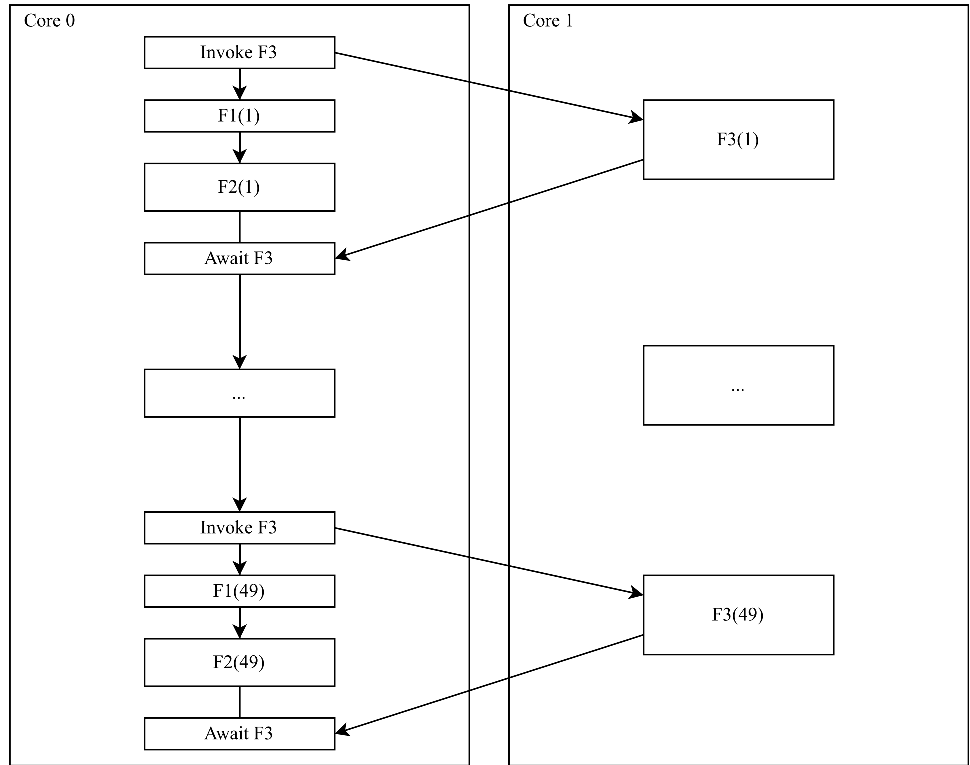

Figure 14 Program flow diagram where F3 is offloaded to a separate core.

Figure 15 Program flow diagram where F2 and F3 are both offloaded to separate cores.

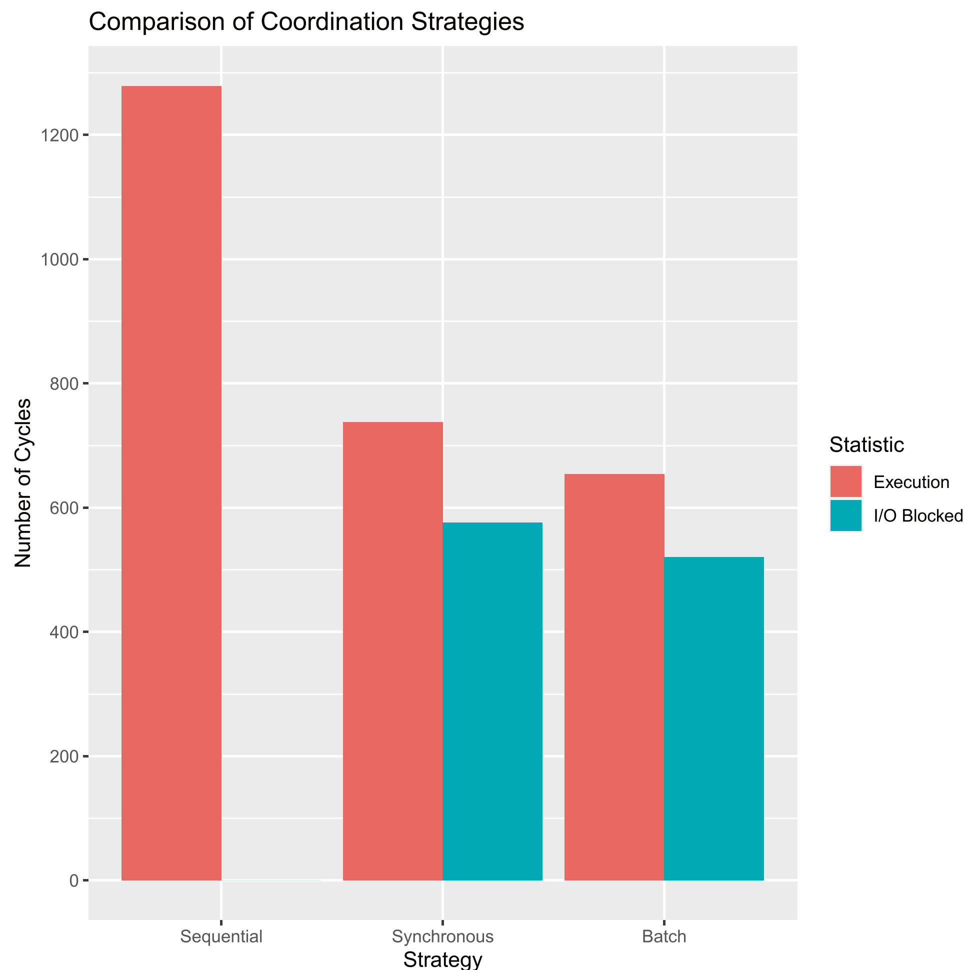

Figure 16 Execution times and number of I/O blocked cycles for each coordination strategy.

coordinator core's call strategy) are very straightforward in IMP-lang.

Figure 17 displays the execution profiles of each of the function allocation strategies. As with the first experiment, the strategy that made no use of parallelism was the slowest to finish, though it also by definition spent no time blocked. As might be expected, offloading F3, the slowest function, provided a large savings in execution time (a nominal 22% savings). Offloading the other functions provided comparatively negligible benefit to execution time, on the order of less than a hundred cycles.

The large number of cycles that the main core spent waiting for execution to complete in the strategy where only F3 was offloaded is consistent with the measured fact that F3 has an extremely outsized computation time compared to the sum of F1 and F2; 63% of the execution time was spent waiting, meaning that only 37% was needed for computing F1 and F2 as well as printing results and issuing commands to the core computing F3. A developer using IMP-lang for design-space exploration could conclude that since even when offloaded, F3 takes up a large amount of time, they could use the time wasted by awaiting its completion differently. By exploring coordination strategies similarly to Experiment 1, they may decide that the extra time could be used for computing iterations of F1 and F2 between each iteration of F3, or on other independent tasks that the system may be used for.

Table 8 Numeric Results of Experiment 2.

Strategy	Execution Cycles	Parallel Cycles	Blocking Cycles
Single Core	21630	N/A	N/A
Offload F3	16803	16792	10604
Offload F3 & F2	16752	16741	14147
Offload F3 & F1	16752	16741	11512

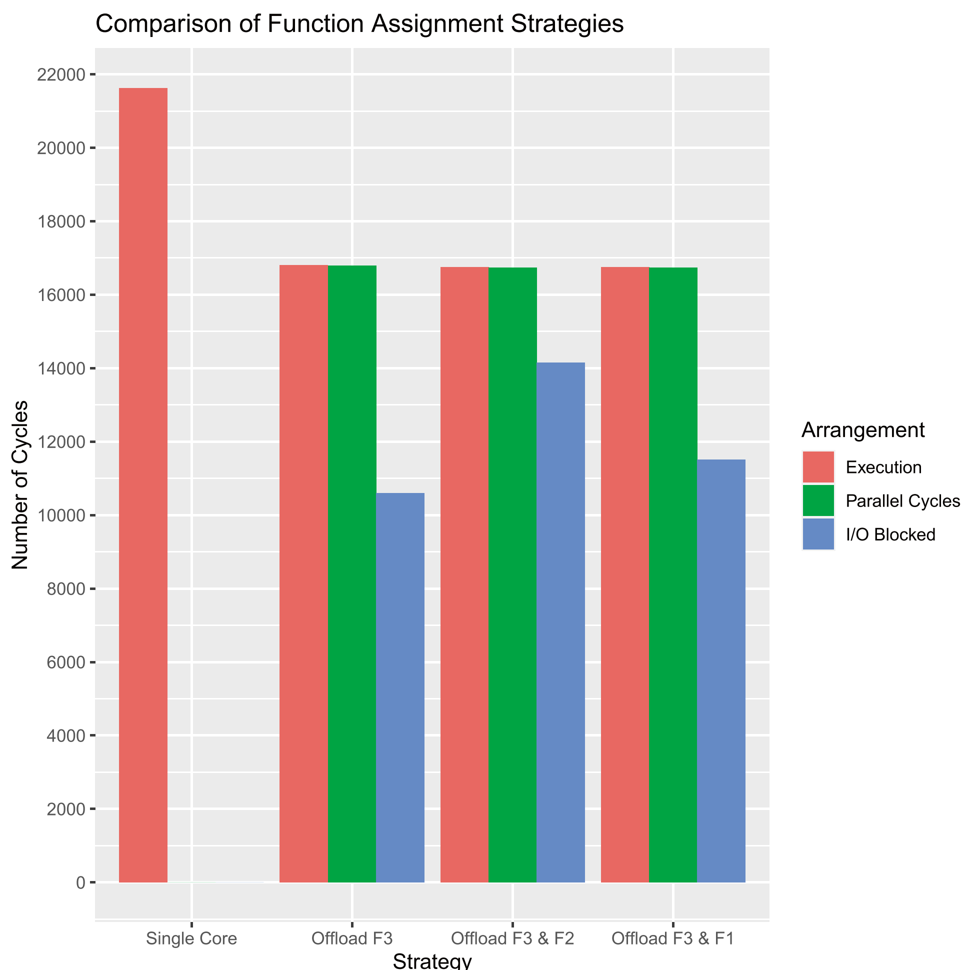

Figure 17 Execution times, amount of parallelism, and number of I/O blocked cycles for each function assignment strategy.

6 Conclusions and Perspectives

This paper offered a primer on hardware acceleration of image processing, focusing on embedded, real-time applications. We surveyed the landscape of High Level Synthesis technologies that are amenable to the domain, and presented our ongoing work on IMP-lang, a language for early stage design of heterogeneous image processing systems.

There are several critical insights readers should take away. First, hardware acceleration is not just a process of converting a piece of computation into an equivalent hardware system: that naive approach offers, in most cases, little benefit. Instead, acceleration must take into account how data is streamed throughout the system, and optimize that streaming accordingly. Second, the choice of tooling plays an important role in the results of acceleration. Different HLS tools, in function of the underlying language paradigm, produce wildly different results across performance, size, and power consumption metrics. Third, as can be observed from IMP-lang, bringing heterogeneous considerations to the language level offers significant advantages to early design estimation, allowing designers to partition their algorithms more efficiently, iterating towards a convergent design that can then be implemented across heterogeneous elements accordingly.

Ongoing and future work must address several challenges. Closer integration of design and development is still required; whilst IMP-lang is a step in that direction, improvements on automatic code generation for different targets must still be performed. Critically, there is still no consensus on the optimum design flow for this class of systems; the methodology and tool landscape is as heterogeneous as the created designs.

Author Contributions J. Fryer was responsible for software development (IMP-Lang) and its description. P. Garcia was responsible for conceptualization and article writing.

Availability of Data and Material Not applicable.

Code Availability Code publicly available under a Creative Commons License.

Declarations

Ethics Approval Not Applicable.

Consent to Participate Not applicable.

Consent for Publication Not applicable.

Conflict of Interest No conflict of interest to report.

References

1. Fu, K.-S., et al. (1976). Pattern recognition and image processing. *IEEE Transactions on Computers, 100*(12), 1336–1346.
2. Chen, Y., Yang, X.-H., Wei, Z., Heidari, A. A., Zheng, N., Li, Z., Chen, H., Hu, H., Zhou, Q., & Guan, Q. (2022). Generative adversarial networks in medical image augmentation: A review. *Computers in Biology and Medicine,* 105382.
3. Salembier, P., & Garrido, L. (2000). Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *IEEE Transactions on Image Processing, 9*(4), 561–576.
4. Abràmoff, M. D., Magalhães, P. J., & Ram, S. J. (2004). Image processing with imagej. *Biophotonics International, 11*(7), 36–42.
5. Bond, J. (1997). The drivers of the information revolution: Cost, computing power, and convergence.
6. Mittal, S., Gupta, S., & Dasgupta, S. (2008). FPGA: An efficient and promising platform for real-time image processing applications. In *National Conference on Research and Development in Hardware Systems (CSI-RDHS).*
7. Huang, L., & Barth, M. (2009). Tightly-coupled lidar and computer vision integration for vehicle detection. In *2009 IEEE Intelligent Vehicles Symposium* (pp. 604–609). IEEE.
8. Brunetti, A., Buongiorno, D., Trotta, G. F., & Bevilacqua, V. (2018). Computer vision and deep learning techniques for pedestrian detection and tracking: A survey. *Neurocomputing, 300*, 17–33.
9. Zhang, X., Chen, Z., Wu, Q. J., Cai, L., Lu, D., & Li, X. (2018). Fast semantic segmentation for scene perception. *IEEE Transactions on Industrial Informatics, 15*(2), 1183–1192.
10. Al-Kaff, A., Martin, D., Garcia, F., de la Escalera, A., & Armingol, J. M. (2018). Survey of computer vision algorithms and applications for unmanned aerial vehicles. *Expert Systems with Applications, 92*, 447–463.
11. Feng, X., Jiang, Y., Yang, X., Du, M., & Li, X. (2019). Computer vision algorithms and hardware implementations: A survey. *Integration, 69*, 309–320.
12. Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience, 2018*.
13. Jinghong, D., Yaling, D., & Kun, L. (2007). Development of image processing system based on DSP and FPGA. In *2007 8th International Conference on Electronic Measurement and Instruments* (pp. 2–791). IEEE.
14. Castaño-Díez, D., Moser, D., Schoenegger, A., Pruggnaller, S., & Frangakis, A. S. (2008). Performance evaluation of image processing algorithms on the GPU. *Journal of Structural Biology, 164*(1), 153–160.
15. Saegusa, T., Maruyama, T., & Yamaguchi, Y. (2008). How fast is an FPGA in image processing? In *2008 International Conference on Field Programmable Logic and Applications* (pp. 77–82). IEEE.
16. Bhowmik, D., Garcia, P., Wallace, A., Stewart, R., & Michaelson, G. (2017). Power efficient dataflow design for a heterogeneous smart camera architecture. In *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)* (p. 8122128). IEEE.
17. Rt-shadows. (2015). Real-time system hardware for agnostic and deterministic OSES within softcore. In *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)* (pp. 1–4). IEEE.
18. Arató, P., Juhász, S., Mann, Z. Á., Orbán, A., & Papp, D. (2003). Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing, 2003* (pp. 197–202). IEEE.
19. Fryer, J., & Garcia, P. (2020). Towards a programming paradigm for reconfigurable computing: Asynchronous graph programming. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (vol. 1, pp. 1721–1728). IEEE.
20. Brebner, G. (1999). Tooling up for reconfigurable system design. In *IEE Colloquium on Reconfigurable Systems (Ref. No. 1999/061)* (pp. 2–1). IET.
21. HajiRassouliha, A., Taberner, A. J., Nash, M. P., & Nielsen, P. M. (2018). Suitability of recent hardware accelerators (DSPS, FPGAS, and GPUS) for computer vision and image processing algorithms. *Signal Processing: Image Communication, 68*, 101–119.
22. Coussy, P., Gajski, D. D., Meredith, M., & Takach, A. (2009). An introduction to high-level synthesis. *IEEE Design & Test of Computers, 26*(4), 8–17.
23. Borkar, A., Hayes, M., & Smith, M. T. (2009). Robust lane detection and tracking with Ransac and Kalman filter. In *2009 16th IEEE International Conference on Image Processing (ICIP)* (pp. 3261–3264). IEEE.
24. Martin, G., & Smith, G. (2009). High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers, 4*, 18–25.
25. Nane, R., Sima, V. M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., & Bertels, K. (2016). A survey and evaluation of FPGA high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, PP*(99), 1–1. https://doi.org/10.1109/TCAD.2015.2513673
26. Trimberger, S. M. (2015). Three ages of FPGAs: a retrospective on the first thirty years of FPGA technology. *Proceedings of the IEEE, 103*(3), 318–331.
27. Meeus, W., Van Beeck, K., Goedemé, T., Meel, J., & Stroobandt, D. (2012). An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems, 16*(3), 31–51.
28. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., & Zhang, Z. (2011). High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 30*(4), 473–491. https://doi.org/10.1109/TCAD.2011.2110592
29. Zhang, X., & Ng, K. W. (2000). A review of high-level synthesis for dynamically reconfigurable FPGAs. *Microprocessors and Microsystems, 24*(4), 199–211. https://doi.org/10.1016/S0141-9331(00)00074-0
30. Compton, K., & Hauck, S. (2002). Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys (csuR), 34*(2), 171–210.
31. Cardoso, J. M., Diniz, P. C., & Weinhardt, M. (2010). Compiling for reconfigurable computing: A survey. *ACM Computing Surveys (CSUR), 42*(4), 13.
32. Lhairech-Lebreton, G., Coussy, P., & Martin, E. (2010). Hierarchical and multiple-clock domain high-level synthesis for low-power design on FPGA. In *2010 International Conference*

on Field Programmable Logic and Applications (pp. 464–468). https://doi.org/10.1109/FPL.2010.94
33. Panda, P. R. (2001). SystemC: A modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on System Synthesis, 2001* (pp. 75–80). IEEE.
34. Loo, S., Wells, B. E., Freije, N., & Kulick, J. (2002). Handel-C for rapid prototyping of VLSI coprocessors for real time systems. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory, 2002* (pp. 6–10). IEEE.
35. Vanmeerbeeck, G., Schaumont, P., Vernalde, S., Engels, M., & Bolsens, I. (2001). Hardware/software partitioning of embedded system in OCAPI-xl. In *Proceedings of the Ninth International Symposium on Hardware/Software Codesign, 2001, CODES 2001* (pp. 30–35). IEEE.
36. Bollaert, T. (2008). Catapult synthesis: A practical introduction to interactive C synthesis. In *High-Level Synthesis* (pp. 29–52). Springer.
37. Feist, T. (2012). Vivado design suite. *White Paper, 5*.
38. Xu, J., Subramanian, N., Alessio, A., & Hauck, S. (2010). Impulse C vs. VHDL for accelerating tomographic reconstruction. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 171–174). IEEE.
39. Cadence. *C-to-Silicon Compiler High-Level Synthesis.* Retrieved November 1, 2022, from https://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf
40. Synopsis. *Synphony C Compiler.* Retrieved November 1, 2022, from https://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/SynphonyC-Compiler.aspx
41. Cadence. *Cynthesizer Solution.* Retrieved November 1, 2022, from http://www.cadence.com/rl/Resources/datasheets/cynthesizer_ds.pdf
42. Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S., & Czajkowski, T. (2011). Legup: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (pp. 33–36). ACM.
43. Mencer, O. (2006). ASC: A stream compiler for computing with FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25*(9), 1603–1617.
44. Nios, I. (2007). *C2h compiler users guide*. Altera.
45. Putnam, A., Bennett, D., Dellinger, E., Mason, J., Sundararajan, P., & Eggers, S. (2008). Chimps: A C-level compilation flow for hybrid CPU-FPGA architectures. In *International Conference on Field Programmable Logic and Applications, 2008, FPL 2008*. IEEE.
46. Villarreal, J., Park, A., Najjar, W., & Halstead, R. (2010). Designing modular hardware accelerators in C with ROCCC 2.0. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 127–134). IEEE.
47. Coussy, P., Lhairech-Lebreton, G., Heller, D., & Martin, E. (2010). Gaut–a free and open source high-level synthesis tool.
48. Tripp, J. L., Gokhale, M. B., & Peterson, K. D. (2007). Trident: From high-level language to hardware circuitry. *Computer, 3*, 28–37.
49. Settle, S. O. (2013). High-performance dynamic programming on FPGAS with OpenCL. In *Proceedings on IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1–6).
50. Fifield, J., Keryell, R., Ratigner, H., Styles, H., & Wu, J. (2016). Optimizing OpenCL applications on Xilinx FPGA. In *Proceedings of the 4th International Workshop on OpenCL* (p. 5). ACM.
51. Papakonstantinou, A., Gururaj, K., Stratton, J. A., Chen, D., Cong, J., & Hwu, W.-M. W. (2009). FCUDA: Enabling efficient compilation of Cuda Kernels onto FPGAs. In *IEEE 7th Symposium on Application Specific Processors, 2009. SASP'09* (pp. 35–42). IEEE.
52. Auerbach, J., Bacon, D. F., Cheng, P., & Rabbah, R. (2010). Lime: A Java-compatible and synthesizable language for heterogeneous architectures. In *ACM Sigplan Notices* (vol. 45, pp. 89–108). ACM.
53. Singh, S., & Greaves, D. (2008). Kiwi: Synthesis of FPGA circuits from parallel programs. In *16th International Symposium On Field-Programmable Custom Computing Machines, 2008. FCCM'08* (pp. 3–12). IEEE.
54. Nane, R., Sima, V.-M., Olivier, B., Meeuws, R., Yankova, Y., & Bertels, K. (2012). Dwarv 2.0: A cosy-based C-to-VHDL hardware compiler. In *2012 22nd International Conference on Field Programmable Logic and Applications (FPL)* (pp. 619–622). IEEE.
55. Pilato, C., & Ferrandi, F. (2013). Bambu: A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–4). IEEE.
56. Kavvadias, N., & Masselos, K. (2015). Source and IR-level optimisations in the hercules high-level synthesis tool. *International Journal of Innovation and Regional Development, 6*(3), 243–266.
57. Harmsen, R. (2012). Compiling recursion to reconfigurable hardware using clash.
58. Li, Y., & Leeser, M. HML: an innovative hardware description language and its translation to VHDL. In *Proceedings of the ASP-DAC'95/CHDL'95/VLSI'95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal* (pp. 691–696). IEEE.
59. Sander, I., Acosta, A., & Jantsch, A. (2009). Hardware design and synthesis in ForSyDe. In *Workshop on Hardware Design Using Functional Languages (HFL 09)*.
60. Singh, S., & Sheeran, M. (2004). Designing FPGA circuits in lava. Unpublished paper. Retrieved October 15, 2022, from https://www.gla.ac.uk/satnam/lava/lava_intro.pdf
61. Hannig, F., Ruckdeschel, H., Dutta, H., & Teich, J. (2008). Paro: Synthesis of hardware accelerators for multi-dimensional dataflow-intensive applications. In *Reconfigurable Computing: Architectures, Tools and Applications* (pp. 287–293). Springer.
62. Hammarberg, J., & Nadjm-Tehrani, S. (2003). Development of safety-critical reconfigurable hardware with Esterel. *Electronic Notes in Theoretical Computer Science, 80*, 219–234.
63. Derrien, S., & Risset, T. (2000). Interfacing compiled FPGA programs: The MMAlpha approach. In *PDPTA*.
64. Aguilar-Pelaez, E., Bayliss, S., Smith, A., Winterstein, F., Ghica, D. R., Thomas, D., & Constantinides, G. A. (2014). Compiling higher order functional programs to composable digital hardware. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 234–234). IEEE.
65. Procter, A., Harrison, W. L., Graves, I., Becchi, M., & Allwein, G. (2015). Semantics driven hardware design, implementation, and verification with rewire. *SIGPLAN Not., 50*(5), 13–11310. https://doi.org/10.1145/2808704.2754970
66. Sharp, R. (2004). 5. high-level synthesis of SAFL. In *Higher-Level Hardware Synthesis* (pp. 65–86). Springer.
67. Sérot, J., & Michaelson, G. (2012). Harnessing parallelism in FPGAs using the hume language. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing* (pp. 27–36). ACM.
68. Hegarty, J., Brunhaver, J., DeVito, Z., Ragan-Kelley, J., Cohen, N., Bell, S., Vasilyev, A., Horowitz, M., & Hanrahan, P. (2014). Darkroom: Compiling high-level image processing code into hardware pipelines.
69. Membarth, R., Reiche, O., Hannig, F., Teich, J., Körner, M., & Eckert, W. (2016). Hipa^{cc}: A domain-specific language and

compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems, 27*(1), 210–224. https://doi.org/10.1109/TPDS.2015.2394802
70. Cuadrado, J. S., & Molina, J. G. (2007). Building domain-specific languages for model-driven development. *IEEE Software, 24*(5), 48–55.
71. Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004* (pp. 75–86). IEEE.
72. Wipliez, M., Roquier, G., & Nezan, J.-F. (2011). Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems, 63*(2), 203–213.
73. Bezati, E., Mattavelli, M., & Janneck, J. W. (2013). High-level synthesis of dataflow programs for signal processing systems. In *2013 8th International Symposium on Image and Signal Processing and Analysis (ISPA)* (pp. 750–754). IEEE.
74. Yviquel, H., Lorence, A., Jerbi, K., Cocherel, G., Sanchez, A., & Raulet, M. (2013). ORCC: Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia* (pp. 863–866). ACM.
75. Hormati, A., Kudlur, M., Mahlke, S., Bacon, D., & Rabbah, R. (2008). Optimus: Efficient realization of streaming applications on FPGAs. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (pp. 41–50). ACM.
76. Thies, W., Karczmarek, M., & Amarasinghe, S. (2002). StreamIt: A language for streaming applications. In *Compiler Construction* (pp. 179–196). Springer.
77. Püschel, M., Moura, J. M., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gačic, A., Voronenko, Y., et al. (2005). Spiral: Code generation for DSP transforms. *Proceedings of the IEEE, 93*(2), 232–275.
78. D'Alberto, P., Milder, P. A., Sandryhaila, A., Franchetti, F., Hoe, J. C., Moura, J. M., Puschel, M., & Johnson, J. R. (2007). Generating FPGA-accelerated DFT libraries. In *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM 2007* (pp. 173–184). IEEE.
79. Stewart, R., Duncan, K., Michaelson, G., Garcia, P., Bhowmik, D., & Wallace, A. (2018). RIPL: A parallel image processing language for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, *11*(1). https://doi.org/10.1145/3180481
80. Nikhil, R. (2004). Bluespec system Verilog: Efficient, correct RTL from high level specifications. In *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04* (pp. 69–70). IEEE.
81. Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., & Asanović, K. (2012). Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference* (pp. 1216–1225). ACM.
82. Synflow. *Introducing Cx*. Retrieved November 1, 2022, from http://cx-lang.org/
83. Edwards, S. A. (2000). Kahn process networks. In *Languages for Digital Embedded Systems* (pp. 189–195). Springer.
84. Fleming, S. T., Beretta, I., Thomas, D. B., Constantinides, G. A., & Ghica, D. R. (2015). PushPush: Seamless integration of hardware and software objects via function calls over AXI. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1–8). https://doi.org/10.1109/FPL.2015.7294024
85. Liu, Y., Bouganis, C.-S., Cheung, P. Y., Leong, P. H., & Motley, S. J. (2006). Hardware efficient architectures for eigenvalue computation. In *Proceedings of the Design Automation & Test in Europe Conference* (vol. 1, pp. 1–6). IEEE.
86. Srivastava, S. (2018). Memory interface design for integrating accelerators with Xilinx Zynq platform.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.