On The Efficiency of FPGA Architecture for Inline Data Transport Computation

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Siddharth Sahay

B.Tech., Computer Science & Engineering, Manipal Institute of Technology

M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University

Pittsburgh, PA

August 2026

Acknowledgments

I would like to thank the large number of people who have offered their support and friendship throughout the PhD process, making it the incredible once-in-a-lifetime experience that it was. Firstly, I would like to thank my advisor, Prof. James C. Hoe, for giving me the freedom to explore my own interests, guided by mentorship on how to think, act, and do research on interesting problems, as well as for creating a supportive environment where I was encouraged to learn new things, participate in a variety of projects, and communicate with other researchers. Many thanks as well to my doctoral committee: Prof. Tze Meng Low, Prof. Jignesh Patel, Dr. Scott Weber, and Dr. Ren Wang. In the many years spent in collaboration with all of you, I have drawn considerable inspiration towards becoming both a better researcher and a better person. I am grateful as well for the feedback and discussion that greatly helped shape and refine this dissertation.

I would like to offer a very warm thanks to the friends and collaborators I made along the way. Thank you to the members of James' research group, past and present: Shashank Obla, Jason Cheung, Eric Tang, William Krska, Simon Yu, Chengyue Wang, Sanjay Seshan, Rock Zhang, Edwin Lim, Li Shi, Joe Melber, Zhipeng Zhao, Guanglin Xu, and others. Additionally, thank you to the members of the A-Level community: Upasana Sridhar, Quentin Oschatz, Larry Tang, Emma Quinn, Elliott Binder, Deepanjali Mishra, Pragna Mamidipaka, Yufei Shi, Ishank Juneja, Sanil Rao, and others. I deeply appreciate the friendship, help, col-

laborations, shared last minute deadlines, discussions on myriad topics, personal inspiration, gaming sessions, and yes, also the cookies (the baking in general). I will remember my time here with particular fondness. Thanks as well to the members of the Crossroads 3D FPGA Research Center and other collaborators over the years, including Martin Prammer and Luiz Gustavo de Sa. I would also like to thank my friends from India and around Pittsburgh, who have offered their unwavering faith throughout.

Additionally, thank you to Mike Kinsner, Joe Garvey, Adel Ejjeh, and Sophie Mao from the Intel-CMU Scythe project, who offered considerable feedback and guidance. This work was funded by CMU CIT Dean's Fellowship, the Intel-VMware Crossroads 3D FPGA Research Center, Intel Corporation (SRS), and DARPA-SOAP.

Finally, I would like to thank my family, especially my parents, for the unending support and affection throughout the process. I would not be here without you.

Siddharth Sahay

Carnegie Mellon University

August 2026

Abstract

With the slowdown in benefits from process scaling, modern computing systems increasingly rely on hardware specialization for performance and efficiency. Simultaneously, data-intensive workloads such as signal processing, low-latency inference, and network security are often bottlenecked by large-scale data movement rather than compute capability. Traditional CPU/GPU architectures and accelerator paradigms typically optimize for high compute density on data resident in deep memory hierarchies, incurring resource and efficiency overheads from the lower arithmetic intensity and limited data reuse in many stream processing tasks. This leads to the challenge of computing efficiently on high-speed streaming data.

This work addresses the gap by proposing specialized FPGA accelerators, which are well-suited for stream processing, integrated along high-traffic data movement paths in modern computing systems. A design study is first presented integrating FPGAs along with GPUs in a signal processing scenario, demonstrating efficiency gains with today's devices used appropriately. However, current FPGAs are optimized for general-purpose workloads, lowering the potential benefit. This is quantified relative to ASIC model, leading to a breakdown of key architectural inefficiencies. These findings are finally used to propose Flow FPGA, a domain-specialized FPGA more suited for efficient stream processing, evaluated using a suite of related tasks. In all, this work demonstrates the suitability of FPGAs for adding inline stream processing to existing computing systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

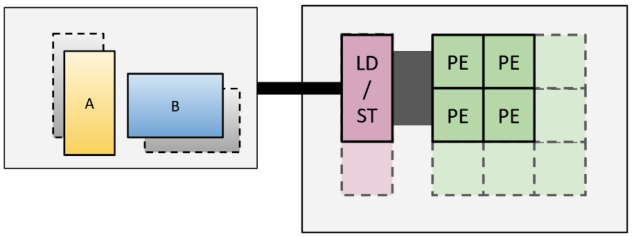
1.1 The Data Transport Computing Problem

At a very high level a computing system can be understood to consist of memory, which stores data; compute, which processes data; and communication, which moves data from place to place such as from memory to compute or between multiple processors [3]. Traditionally, computing systems have separated these three categories, with CPUs, for example, being a compute device that acts on data from DRAM memory over a memory communication channel. However, in modern applications such as data analytics or high-bandwidth radar processing, which frequently involve large data bandwidths, this separation causes a large amount of energy to be expended simply moving data into the compute node [4]. As will be seen later, raw compute also consumes less power than memory or communication in terms of energy per bit [5]. This is more apparent with the end of easy improvements from Moore's Law with Dennard scaling [6, 7]. Therefore, if the compute cannot reuse a data block multiple times to amortize its data storage and movement costs, the energy will be dominated by memory and communication [8]. This is especially the

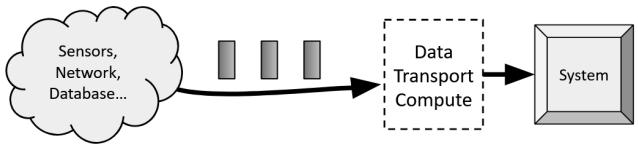
case for computations with low data reuse, which by definition only perform a small number of operations on each data block [9].

The memory and communications stack of a typical modern CPU or GPU is optimized for high data reuse of data held close by in DRAM through an extensive cache hierarchy, leading to a large initial data storage and movement cost per bit, but cheap re-access from temporal and spatial locality [1]. For such a system, low-reuse computations will lead to inefficient implementation, where data travels up a complex storage and movement structure only to be used a few times [10]. Additionally, many high-bandwidth computations have data-reducing or filtering stages, where only a fraction of the data is actually used in further computation. This nevertheless requires transferring the entire input bandwidth to the computing device only for a large fraction to be filtered [11]. This loss in efficiency increases the amount of energy required for such computations, which is detrimental to power and cooling requirements, especially on constrained or large-scale deployments such as mobile platforms and data centers [12].

High-Level Problem Overview. To mitigate this, computer architecture research and practice has explored pushing compute into memory and communications. The efforts around Processing in Memory for example solve this problem by integrating computation at or near memory systems, removing or mitigating the need for the trip to a central compute node [13, 14]. In this dissertation, the analogous problem of computing in the communications domain [15] is explored for the specific case of performing streaming compute on data that is in transit from an upstream source, such as a sensor or a network, to a downstream endpoint, such as a CPU or a GPU, as seen in Figure 1.1. **In this work, this is termed the "data transport compute" problem, as it deals with computing on data as it**



(a) Classical view of computation where data in memory is moved to compute units over a communication link.



(b) Data transport compute performing inline operations on the compute path.

Figure 1.1: Data transport computing scenario.

is transported to an endpoint to reduce bandwidth or offload low-reuse operations, with the goal being to provide a feasible, flexible, and most importantly energy-efficient solution. A particular feature of this problem is that the upstream source generates and sends data regardless of downstream i.e. there is no flow, ordering, or access pattern control to upstream, the data stream as received must be handled at all costs. This is the case with sensors and network data sources which fall into the supported domain, as opposed to closer-by sources such as DRAM or local storage where flow, ordering, and access pattern control is both common. As an additional note, this is related to but distinct from existing work on bump-in-the-wire acceleration [16, 17, 18], as that does not restrict the computation modality and often involves infrastructure processing work. This will be further substantiated in Chapter 2.

In the identified applications such as data analytics and adaptive beamforming, a common archetype is that of a simple feed-forward processing pipeline, where

streaming data is ingested, gets computed upon, then exits the system also in a streaming fashion. For example, in data analytics, a group-by aggregation operation performed on the communication path as data is being transferred to the CPU would take as input a stream of key-value pairs, groups the values by the keys and aggregates each group through a streaming associative insert and lookup, eventually emitting the aggregated values [19, 18]. Similarly, in beamforming, a streaming polyphase filter is used to transform the input signal into frequency bands, from where only the required bands can be selected [20]. This archetype places several important restrictions for an efficient implementation, namely (1) imposing an external schedule on the operations dictated by the input data stream, (2) bounding the reuse and latency to what is required for line-rate processing, and (3) avoiding DRAM-class memory as reads and writes to that are expensive operations and moreover not required due to the bounded latency as long as staging and pipelining buffers fit in on-chip memory. It also dictates interface considerations, as the natural interface is a single streaming input pipe and a streaming output pipe, however many systems such as CPUs and GPUs do not support this direct streaming interface. Thus, there is both an interface and a computation challenged to be solved here.

Properties of the Solution. In this work, efficiency (at required performance) is the primary factor considered, measured in terms of operations per watt for handling the input rate. All the applications in this dissertation are assumed to be operating at steady state, or close enough that the average power is as good a metric as total joules in terms of effectively defining system efficiency. However, efficiency on its own would simply result in an Application Specific Integrated Circuit (ASIC) every time, as bespoke hardware built exactly to purpose can optimize to an arbitrary

degree limited only by the process node [8, 21]. This results in a very expensive and non-reusable solution to the problem that incurs a large amount of time, engineering effort, and development cost [22].

Therefore, the solution sought has a secondary constraint, which is flexibility through programmability [23]. Here, flexibility is defined in the sense of being able to combine processing blocks in novel ways as long as they still conform to the overall shape of a data transport computing problem, and to be able to implement occasional operators or functionalities that are not covered by existing processing blocks. Adding flexibility to a generalized ASIC in this way would require adding programmable control, sequencing, and data routing to the ASIC's processing blocks, as well as a mechanism for implementing new operators.

Thesis Statement. From the above discussion and requirements, Field Programmable Gate Arrays (FPGAs) are identified as a good solution to the data transport computing problem. As substantiated in Chapter 2, a modern FPGA has flexible streaming-capable IO, a large number of hard blocks for efficiency, and programmable soft logic and routing allowing for novel combinations of the hard blocks as well as operator flexibility [24, 25]. The computational archetype readily maps to an FPGA's streaming IO and internal data flow consisting of soft logic and hard blocks. However, FPGAs today are optimized for general-purpose use, or for very special domains like ASIC emulation or telecommunications, and not for the computational archetype for data transport computing [26]. This suggests an approach of optimizing the FPGA's design tradeoffs towards this domain for additional efficiency, recovering some of the lost ground to ASICs and therefore making ASICs an increasingly unappealing solution as the engineering effort results in only a somewhat more efficient device, without any of the FPGA's flexibility [27, 28].

This thesis therefore seeks to demonstrate that (1) today's FPGAs, used well, can solve the data transport computing interface and computation challenges feasibly and at a higher efficiency than a traditional accelerator such as a GPU, and (2) an FPGA with domain-specific architectural optimization can significantly close the gap to ASICs, providing an efficient, feasible, and flexible solution to the problem.

1.2 Example: Accumulating Streaming Integers

Consider a very simple example problem of accumulating a stream of integers arriving from a live data source such as a sensor or being read from a remote database table. A simple extension of this problem to key-value pairs results in group-by aggregation, one of the example kernels. Typically, one would only consider the computational cost of this operation, which, for example, on a common Central Processing Unit (CPU) might be expressed as a simple for-loop that reads data from the input source and accumulates it into a register. However, this misses the path that such a stream must actually take to reach a CPU's functional unit. This detailed and extensive path is documented below to demonstrate that the CPU is simply not designed for data transport computing operations, as the majority of the energy goes to data movement rather than compute. Even with modern optimizations such as kernel bypass networking [29, 30], a significant number of steps are required for a data bit to reach the CPU's arithmetic units. In contrast, using an FPGA allows for a direct path from the input to a hard block in accumulation mode over the programmable routing, resulting in a much lower energy per bit [24]. This underscores the FPGA's architectural efficiency at this problem domain.

Stage (pJ / 64 b datum)	CPU	FPGA
Ethernet SERDES/PHY rx (long reach) [31]	300	300
PCIe Gen5 link, NIC→host [32]	730	—
DMA→DRAM write [5, 33]	1300–2600	—
Kernel→user copy [5]	2600–5200	—
LLC read [5]	100	—
L1 load [5]	20	—
Instr. overhead [5]	210	—
2×32 b adds (work) [5]	0.2	—
Fabric routing [34]	—	100
Pipeline registers (Ch. 4)	—	5
Compute block accumulate (Ch. 4)	—	10
Total	≈5300–9200	≈420
Total (DPDK+DDIO)	≈1500	≈420

Table 1.1: Streaming accumulation energy per 64b datum. Several entries derived from cited sources.

For these examples, it is assumed that the data come through a common network interface over a simple protocol such as Ethernet. **Additionally, while the energy of the path has been quantified using numbers directly from or derived from literature, in practice it is difficult to do so accurately due to the complexity of modern systems, and therefore this argument should be taken more qualitatively than quantitatively.**

Generic CPU Solution. The generic CPU case with default drivers and commonly used networking stack has a particularly expensive energy cost and is illustrated in Figure 1.2. The stream would be read in by a receiving unit on the CPU side, which is assumed here to be an Ethernet physical layer and serializer-deserializer (serdes) that interfaces with the raw Ethernet signals on the wire. The Ethernet network interface (NIC) therefore performs the physical operations to accept the actual data packets, then writes them to kernel buffers on the DRAM,

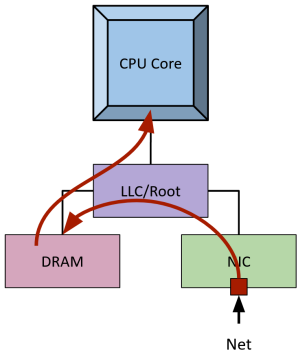


Figure 1.2: The CPU data streaming path.

finally alerting the CPU to the receipt of new packets using a notification mechanism. The CPU then reads those buffers, typically first transferring them from the NIC kernel buffers into a sequence of operating system and user buffers depending on the networking stack, with every buffer transfer requiring cache updates if not actual DRAM reads and writes. Due to the universality of virtual memory on CPU systems, any pointer operations involving buffers must energize a TLB lookup (translation lookaside buffer) for the address translation in parallel with the cache lookup. Finally, the CPU would load the packet into its last level cache and the data would make its way up the cache hierarchy, passing complex miss-handling registers, cache partitioning and allocation, coherency permissions, and other relatively complex mechanisms to eventually make its way into the L1 data cache. From here, a single unit of that data would be read into a CPU register by the load instruction that triggered this sequence of events in the first place (passing through the instruction fetch, instruction cache, the CPU frontend, reorder buffer, load-store queue, etc.) into a register, where upon being allocated an execution unit, the add operation is done [1]. All these factors contribute to the large number of layers and the large energy per word in Table 1.1, at about 5300-9200 pJ/64b.

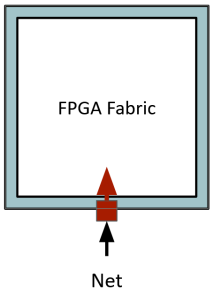


Figure 1.3: The FPGA data streaming path.

Optimized CPU Solution. Modern networking stacks such as Data Plane Development Kit (DPDK) [30] along with features such as Intel Data Direct IO (DDIO) [35] **allow for the bypassing of several of the above buffering stages, resulting in both more performant and efficient implementation.** This is done precisely to remove some of the overhead mentioned above. However, even with kernel bypass networking, the data still crosses a variety of hardware layers such as PCIe, LLC, and each caching layer. Kernel bypass networking reduces or removes the software contribution to the energy floor, but it cannot remove the hardware contribution. This removes some of the intermediate layers (especially multiple passes through DRAM). With some optimistic assumptions, this can lower the cost to "only" approximately 1500 pJ/64b.

This data movement path is the result of three key architectural choices behind the CPU, namely, (1) the requirement for instruction-programmability (the instruction path and the extensive automatic structures), (2) memory-like interface to the rest of the system (where the NIC has to interface with the system over memory), and (3) the optimization for locality (the extensive and automatic cache). None of these are relevant to data transport computing.

Simple and Efficient FPGA Solution. In contrast, on an FPGA the solution is much simpler, as shown in Figure 1.3. The incoming data is read into a data stream output via a FIFO-like interface. This can then be routed more or less directly to a nearby compute block where the integers can be accumulated using the accumulation mode. The entire path consists of only a few wires and internal clocked pipelining registers instead of the extensive network of caches and other complex automatic machinery in the other examples. Therefore, for this sort of problem, the FPGA is simply a better natural fit architecturally. **A simple accounting from Table 1.1 puts the FPGA at only 420 pJ/64b compared to the CPU's 9200 pJ/64b or the optimistic 1500 pJ/64b.** Note that the Ethernet MAC has been omitted on the FPGA side for lack of a reference, but its effect will be minimal at best as it is a hardened block on modern FPGAs [24].

1.3 Difficulty on Existing Architectures

A short discussion is provided below on why most existing architectures do not perform either feasibly or efficiently for the data transport computing problem.

Computational Mismatch. Most common accelerator types, including GPUs, TPUs, and many NPUs, are built for high-throughput high-reuse computing, as that is the requirement in their deployed applications such as AI training and inference. For example, a GPU has an extensive multi-level caching structure and a block memory IO system that is optimized for reading and writing large blocks of contiguous data all at once [36, 37]. This maps well for a kernel such as a large tiled dense matrix multiplication, as large cache blocks can be fetched, stored, and reused efficiently among the GPU's large number of parallel compute units, assum-

ing a good kernel implementation. Similarly, accelerators such as Tensor Processing Units (TPUs) and many similar such are systolic array structures with software-managed scratchpads with deeply pipelined memory units and a geometry hardwired for this domain that expect high-reuse computations with deep control over data access [38, 39]. While data transport computing kernels may be mapped to these with appropriate performance, they cannot be mapped with appropriate efficiency.

Control and Interface Mismatch Matching data residency, GPUs follow a *compute-offload* model [40], where the host CPU sends a block of work items over PCIe to GPU device memory, as well as invokes a compute program, eventually pulling the results back over PCIe. Along this data movement path, there are multiple microarchitectural structures involved, such as the last-level cache (on both sides) and DMA units. Modern GPUs do have the ability to accept data instead over RDMA over the network [41] or via cache coherent NVLink-C2C [2], but even there data and work is transferred using a block-memory modality (under automatic caching that is at best hinted at). For generic streaming workloads where the data is coming in directly from, for example, a network or a sensor, this requires intermediate devices (such as a SmartNIC) to convert into the GPU's memory-like interface, often incurring a trip through GPU DRAM in the process. This again is a consequence of the high-reuse data-resident centric problem domain that these devices are built to solve.

Accelerator Landscape, Briefly. There are other accelerator architectures which solve problems similar to or overlapping with data transport computing, which are briefly enumerated here. A fuller treatment is provided in Chapter 5. Firstly, the most obvious comparisons are DPUs and SmartNICs such as NVIDIA BlueField [42]

and Intel IPU family [43], which handle streaming workloads in some form or the other. These networking oriented devices are built for infrastructure acceleration instead of compute, and often feature manycore CPUs which, while doubtless more optimized than generic CPUs, still suffer from the same architectural weakness explained above. FPGA SmartNICs such as Silicom N6010 [44] use off-the-shelf fabrics and do not optimize as this dissertation does, however, this is by far the closest market comparison (differentiated by architectural tradeoff for the chosen application space). Network computing devices such as Tofino [45] or other P4-programmable devices follow a limited match-action abstraction, which is limiting for compute purposes. Streaming dataflow accelerators from AI acceleration vendors such as Groq [46], SambaNova [47], Tenstorrent [48], and Cerebras [49] do allow deterministic streaming, but are rather closer to the TPU case of computing on memory resident operands and are instead optimized for compute-based ML, both of which are different from the data transport case. Therefore, in the current landscape, no architecture quite handles the kind of bandwidth-bound efficient low-compute processing that is required in this dissertation.

1.4 FPGAs for Data Transport Computing

In this dissertation, Field-Programmable Gate Arrays (FPGAs) have been identified as a suitable vehicle to solve the data transport computing problem efficiently. FPGAs are programmable spatial accelerators that have typically been seen as lower cost replacements for Application-Specific Integrated Circuits (ASIC), that is, purpose-built hardware, which pay a relatively hefty price in performance and efficiency for their particular brand of "hardware" programmability [27]. This has classically limited their use for only certain cases where ASIC-like behavior was re-

quired, but the overhead of the FPGA was acceptable. A more extensive background on FPGA structure, interface, and programming is provided in Chapter 2.

Modern FPGAs are Streaming Accelerators. However, the modern FPGA fabric is an elaborate spatial acceleration architecture with hardened compute and memory blocks, allowing for unexpectedly high performance and efficiency when used well. Moreover, the ability of their internal routing architecture to route data directly from where it is generated to where it is used, without requiring a memory or cache-like structure, allows it to very efficiently stream data directly from inputs to computation to outputs, exactly the kind of structure appropriate for data transport computing [24]. Additionally, FPGAs allow for complete customization of integrated interface, starting from very basic generic serdes (serial) interfaces that transfer either raw bits or according to a protocol like PCIe, to wide parallel DDR (double data rate DRAM) and HBM (High-Bandwidth Memory) interfaces [50], allowing for all manner of flexible integration. For a pure streaming workload, the FPGA allows one to simply use a direct serial or wide parallel interface with no overhead from a complex protocol or additional functionality. In all, this makes them well suited to the problem at both the (micro)architecture and the interface level.

FPGA Limitations. FPGAs have several disadvantages that must be acknowledged here. ASICs, GPUs, and other accelerators can often easily go up to multi-gigahertz clock rates, decreasing the required hardware for a given problem size (as they can do more with less given the higher speeds, modulo the cost of additional pipelining stages and high-speed RTL design) [27]. Additionally, FPGAs are relatively difficult to program and have a longer bringup and debug cycle than GPUs. However, this is not a criterion optimized for in this work, and indeed, the benefits

of FPGA programming (bit-level programmable flexibility) are assumed to offset that difficulty, as creating an ASIC would also require a bespoke programming environment. Additionally, ASIC construction is a longer and riskier process given the fixed nature of the device and the multiple hardware spins involved.

Why FPGAs Now? The case for FPGAs providing streaming compute is not new, as prior work has demonstrated bump-in-the-wire FPGAs for data bandwidth reduction, among other things [16, 18]. However, data rates have gone up considerably in the modern era, with 800G line rates outgrowing CPU capability, the end of free upgrades from Moore's Law and Dennard scaling [6, 7] making per-byte energy the key power constraint. With modern FPGAs having a wide selection of hard blocks and rich data interfaces, they are eminently suited for this domain. Together, these trends push classical architectures out of the feasibility zone for data transport compute, while pushing FPGAs further into it. Moreover, package integration today is relatively simple with chiplet and 3D integration allowing the easy assembly of heterogeneous devices, making the system integration easier [51].

However, FPGAs are tuned for general-purpose usage and there is room to tune the architecture to suit the particular needs of data transport computation better, allowing them to get closer to ASIC-tier performance. As the development and manufacturing of ASICs is a difficult and more importantly very expensive endeavor, each step this tuned FPGA can take towards closing the gap with ASICs allows it to displace the requirement of an ASIC from an increasingly larger number of deployment scenarios [22]. Additionally, the programming of ASICs requires a custom programming interface to be developed and verified as well, while FPGA bit-level programmability comes "for free" as long as the fabric is parameterized correctly (and expressed in a standard tool such as VTR [52]). This reduces the

total cost of ownership to solve this problem in a modern system.

FPGA With Data Movement Optimization. An important aside is that there are already FPGAs with data movement structures, such as the NoC (network-on-chip) enabled AMD Versal [53, 54], Altera Agilex M-series [55], Achronix Speedster 7t [56], and academic projects like RAD-sim [57]. On the surface, this is already well suited for data transport compute. However, the role of a NoC is orthogonal to this effort and indeed only supports it, as a NoC is meant for long-range efficient data transport inside the fabric, and does not solve the structural issues of actually computing on the data [58, 59]. Moreover, many of the NoCs present in the above devices are optimized for load-store memory interfaces with request-response traffic, instead of direct streaming, adding unnecessary overhead. However, a good data transport compute device could very well utilize a NoC for long-range data transfer on a larger fabric instance (such as hosting this work in RAD-Sim), which is orthogonal to the problem solved here.

An additional comparison point is to a CGRA such as the Versal AIE (AI Engine) [60, 54]. This again does not have the bit-level flexibility of an FPGA, and incurs overheads through its instruction driven architecture. Additionally, its coarse grained architecture limits problem sizes as for example the streaming width between AIE tile cannot be changed. Additionally, it is over-provisioned in compute for the given applications, being optimized for ML as well as DSP, whereas the architecture being looked for here is for a bounded class of compute problems where many typically have extremely low actual compute.

Application	Data Source	Computation	Destination
Beamforming	Radar	Radar DSP	GPU
Analytics	Database	Aggregation	CPU
Intrusion Detection	Network	Rule Matching	CPU
ML Inference	Particle Accelerator	KAN Inference	System

Table 1.2: Application domains and mapping to data transport computation.

1.5 Data Transport Compute in the Real World

In this dissertation, motivating kernels from the following applications will be used. The primary application domain is the first, beamforming, with the rest providing kernels and analysis material, listed in Table 1.2.

Adaptive Beamforming for High-Bandwidth Signals. Beamforming is a common signal processing operation in applications such as radar, where a number of receiving elements placed in a known geometry (such as a grid) can be used to isolate a signal from a particular direction, while rejecting noise and interference from other directions. The particulars of this problem were given by the DARPA Scalable On-Array Processing (SOAP) challenge. In this problem, each radar element in an array emits a stream of samples at a rate of 40 gigabits per second (40G), which over a thousand such elements leads to a total input bandwidth of 40.96 terabits per second. Computationally, this consists of applying a class of standard algorithms known as Minimum-Variance Distortionless Response (MVDR) [61], which computes a set of weights per element and then combines the weights together using an inner product. The weight computation involves covariance and other algorithms that map well to a GPU. However, the beamforming section including

analysis and synthesis filters as well as inner products has low data reuse and high stream throughput throughout, making them data transport kernels. The proposed system is thus a hybrid FPGA+GPU system, where the weight computation has been mapped to the GPU and the FPGA is integrated along the data movement path, exactly the right shape for a data transport computation study covering both efficiency of kernels and system integration effects. This will be the subject of a detailed design study in Chapter 3.

Group-By Aggregation. In data analytics workloads, aggregation is a key query operator that runs in most analytics queries [62], as visible in the TPC-H collection of benchmarks [63]. As the data is often stored in data warehouses or other storage architectures that are separate from the place where the data is processed, the data is often streamed over the network [64]. The input stream consists of key-value pairs, where the objective is to aggregate all values belonging to the same key with a simple aggregation operator, such as summation or counting. This is repeated for every unique key [18]. The most important operation here is the associative insert and lookup, as the actual accumulation (a direct extension of the addition example above) is a very low arithmetic intensity operation [65]. Therefore, this follows the data transport computing template, but with an added associative operation, highlighting the non-numerical nature of the workload as an added differentiator. This will be solved through specialization.

Network Intrusion Detection. Intrusion Detection Systems (IDS) in a networking context are security appliances that scan network traffic for malicious patterns. A rule-based IDS/IPS has a list of regex rules against which the packet stream must be checked, with successful matches flagging packets as potentially suspicious. This

is a large string matching workload that needs to be done at the networking line rate to keep up with the traffic. Pigasus IDS/IPS [66, 67] is a direct example of a data transport computation workload that is more *performant* on FPGA than on CPU, as evidenced by the original work's ability to substitute the CPU work of multiple servers with a single FPGA implementation. This also implies that it is already more *efficient* as well, with a quotable 38x less power than a CPU-only approach. In this dissertation therefore, Pigasus is presented as an existing example of the kind of acceleration being analyzed. Concretely, a Pigasus stage will be used as an example kernel in a later chapter, with an analysis done for considerations due to the rest of the design.

ML Inference A recent work, KANELÉ [68] presented a method of LUT-based FPGA inference of ML models, where the inference is carried out not by computing weights and activations in compute blocks, but by looking up values in FPGA lookup tables (LUTs), which is the exact operation for which these LUTs are designed. This method involves the use of Kolmogorov-Arnold Networks (KAN) to design ML models that take direct advantage of FPGA fabric. This has suggested applications in ML-based filters for very low latency and high throughput applications such as classifying the outputs of particle accelerators, as these inferences can be made in nanoseconds using FPGA architecture [69]. While still at relatively simple networks, a proof-of-concept KAN was ported for this study (after verification on real hardware). The inference flow in KANELÉ also flows unidirectionally along a data transport path, making this a data transport compute problem. This is a strange example in that the "hard specialization" is actually tailored for FPGA soft logic, as that is the required lookup table and routing structure. Therefore, it acts as a counterbalance for data transport compute shaped computation which is

entirely in soft logic, and therefore acts as a good test for any geometry, IO, and system integration changes.

1.6 Contributions and Structure

This dissertation contributes the following studies and artifacts.

1. **Chapter 3.** Design study of heterogeneous adaptive beamformer including kernels, system-level effects, power, and a characterization of the IO and compute capability of current FPGAs based on this application.

2. **Chapter 4.** ASIC baseline model constructed from literature surveys and operator studies, comparing a breakdown of component power to an equivalent breakdown on the FPGA side, to enumerate the areas of FPGA inefficiency.

3. **Chapter 5.** Flow FPGA, an optimized fabric for data transport workloads defined as a thin FPGA-like compute strip sandwiched between wide parallel IO interfaces (a "compute stop for data in transit"). Defined a resource model deriving fabric composition from transport problem characteristic, a VTR-based toolchain [52] for Flow FPGA fabric, and results from design sweeps for the studied kernels.

4. **Chapter 6.** A qualitative evaluation of system integration considerations for the proposed Flow FPGA, covering logical and physical integration.

The goals of each chapter are summarized below.

Chapter 2: FPGA and Data Movement Background. This chapter provides common background material on FPGA fabrics, programming methodology,

toolchains, and FPGA IO and data movement options as available in today's devices. It also surveys prior work on using FPGAs for inline computing. Additional background material is provided in the following chapters as necessary.

Chapter 3: Data Transport-Centric Design Study for a Digital Beamformer. In this study, a system design is carried out for an accelerated digital beamforming platform according to the DARPA SOAP challenge application identified above. Radar DSP details and the other goals of the SOAP challenge are not in scope here, as the primary goals of this study are to demonstrate, using a current-generation FPGA device, the first claim of the thesis, that FPGAs are already good solutions to the identified data transport computing problem. This study focuses on both the above criteria, namely the efficiency of transport kernels when implemented on FPGAs, as well as the system interface and integration solutions offered by modern FPGAs with high-performance 100G transceivers and HBM interfaces. Finally, this study closes by identifying some of the architectural tradeoffs made in today's devices exposed by scaling up the problem to larger instance sizes, such as the provisioning of compute vs IO, providing data for the optimization effort below.

Chapter 4: The ASIC Gap, Quantified. The previous study was a "one-shot" solution to a particular problem. While the trends in FPGA design tradeoffs are a useful starting point (such as increasing the total IO), any optmization effort is left rudderless without a nuanced baseline to compare the various aspects of the FPGA to, such as the compute, routing, soft logic, clocking, etc. For example, it is not possible to reason about whether the optimization effort should focus on making the compute blocks more efficient, or the data movment more efficient, without comparing a baseline ideal on each of those components. As no pre-existing ASIC

currently exists with a detailed published breakdown of these sub-components for exactly this given problem, this study then constructs this baseline through a systematic methodology of replacing functional blocks and data routing with ASIC blocks synthesized using an open source toolchain. Along with detailed power estimates from the FPGA EDA tools, this allows for a nuanced comparison on where the FPGA power is actually being wasted. It is important to note that the objective of this baseline study is not to construct an actual working ASIC, but rather a useful comparison point. It is used to establish a more objective baseline for the Chapter 3 results, and be a comparison point for Chapter 5.

Chapter 5: Flow FPGA: Data Transport Optimized FPGA. This study builds on all the information that has been gathered before this, to finally propose a new data transport-oriented family of FPGA architectures, termed Flow FPGA. Flow FPGAs are tall and thin devices with large IO shorelines supporting many parallel data flows, with a customized hard-block-centric fabric for maximum efficiency. This removes excess logic, wiring, and distance to IO that is not required in the identified application. A new mix of hard blocks is prototyped covering the applications studied, showing that after appropriate hard block usage, the fabric left over is easily expressible in the much more limited Flow soft logic. The fabric still retains some flexibility from still being FPGA and still being VTR-programmable, allowing for the limited implementation of new operators as long as they fit the same mold. The results of this study will be a design space of new FPGA fabrics, which are optimized for data transport computing, evaluated against the Chapter 3 results and the Chapter 4 baseline.

Chapter 6: System Integration. This chapter is speculative in nature and carries a largely qualitative discussion of various system design and integration considerations that are enabled due to the presence of a Flow FPGA, such as physical interfaces, logical interfaces, and host-side considerations.

Chapter 7: Conclusion and Future Work. This chapter summarizes this dissertation, places it in context among the larger trends in computer architecture research, and suggests new avenues of research that have been enabled through this work.

1.7 Limitations

This is an architectural study, not performed at the devices and circuits level. Therefore, the Flow FPGA has not been validated on real silicon or with a circuit-level implementation, but with performance and power models as well as a mock-up in VTR, a standard academic FPGA toolchain [52]. Large parts of the analysis have been carried out with a relatively small number of application domains, with results assembled in a "pen-and-paper" fashion instead of through assembling and testing a full system. While effort has been taken to keep these results realistic, it is entirely possible, even probable, for there to be real system effects not accounted for (however, the conclusions likely do not change).

Chapter 2

Background

2.1 Field-Programmable Gate Arrays

FPGAs Classic To Modern. A Field-Programmable Gate Array (FPGA) is a VLSI device historically consisting of a programmable fabric with configurable look-up tables (LUTs) and programmable routing of wires and switches. FPGAs can be used to implement any desired digital circuit through suitable configuration, a process typically much more efficient than emulating the circuit in software. However, this carries some resource and performance loss compared to implement-

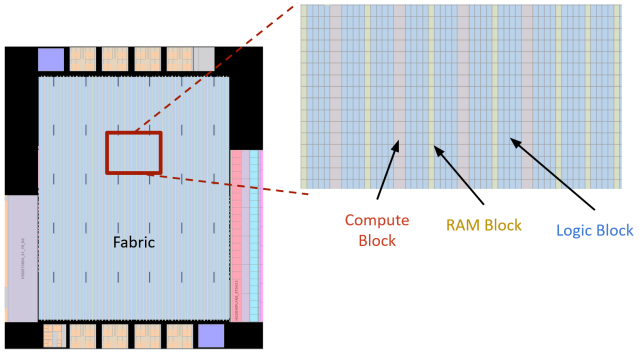


Figure 2.1: Modern FPGA showing compute, memory, and logic blocks.

ing digital circuits directly in silicon, notably an increase in area and a decrease in maximum frequency from using programmable structures instead of purpose-built ones. Their primary usage is in implementing hardware designs without incurring the cost and development time of an Application-Specific Integrated Circuit (ASIC) solution where such tradeoffs are acceptable [26].

Device	**Logic Elements**	**DSP Blocks**	**M20K Blocks**
AGF 012 / 014	1,437,240	4,510	7,110
AGF 022 / 023	2,296,550	7,204	11,356
AGF 027 / AGI 027	2,692,760	8,528	13,272
AGI 035	3,540,000	9,594	14,931
AGI 040 / 041	4,047,400	12,792	19,908

Table 2.1: Compute and memory specifications of a modern Altera Agilex 7 FPGA.

To improve performance of the implemented designs, FPGA vendors over the years have hardened commonly used capabilities, such as numerical operations, memory, data transport, and IO as shown in Table 2.1, along with improvements towards improving the maximum frequency of FPGA designs, such as embedding dedicated pipeline registers. Therefore, the modern FPGA is a sophisticated platform consisting of programmable soft logic, various types of configurable high-performance hard blocks, and a plethora of high-speed data transport options [24], as shown in Figure 2.1. This can be treated as an acceleration platform in its own right, and not simply as a replacement for an ASIC.

For complex and floating point arithmetic, multiple compute blocks may be needed at a given time to implement a single operation, such as 4 for a complex float32 multiplication. Apart from this, FPGAs can also implement compute and memory in LUTs, and may do so even in high frequency designs when it is easier to route than adding another set of hard blocks. These devices can relatively easily be

clocked at 500MHz+, but because of modern architectural innovations like HyperFlex [70], which embeds dedicated pipelining registers in the fabric, carefully made designs on high speed grade devices can run at up to 900+ MHz [71].

The trend of incorporating hardened blocks has also led to vendors creating specialized FPGAs for various domains to cater to different markets, such as very large compute-less FPGAs for simulation, or FPGAs with tensor blocks for AI [72]. However, these are still treated as ASIC replacements. A primary outcome of this thesis, therefore, is to identify one such important niche for FPGAs, and, through a combination of fabric optimization and contextualizing performance within a full-system application, prove that a congruently designed FPGA, used well, is sufficient for solving the computing problems within that niche, eliminating the need for developing a custom ASIC solution.

Suitability for Streaming Computation. FPGAs are particularly adept at data flow and streaming patterns, due to the architectural feature of being able to directly route data to and from computation hard blocks, unlike traditional architectures where data routing must go through memory or the internal dataflow of a processor. This design pattern also moves the programmable soft logic, usually a significant component of the overhead and inefficiency of an FPGA design, into a more control and supervisory role, being used to implement state machines, arbitration, and other secondary functions. Additionally, FPGAs have robust IO options in the form of a large number of GPIO pins and transceiver blocks, allowing them to directly ingest streaming data without the overhead of buffering in DRAM first [73]. Due to these factors, FPGAs can attain very high performance and very high efficiency for streaming assuming the computation fits. Additionally, because the IO overhead is small and that FPGA can control exactly the resources used along the

computation path, decreasing the arithmetic intensity of the streaming computation has a much smaller effect on the FPGA than it does for CPUs and GPUs, leading to a very large advantage in both efficiency and performance in applications with low arithmetic intensity [74].

2.2 FPGA Programming and HLS

Manual Programming through RTL. Modern FPGA CAD tools ingest designs written in register transfer languages (RTLs) such as Verilog and VHDL, and through synthesis and place-and-route operations in a manner similar to ASIC design tools, convert them into bitstreams for programming FPGAs. Through the use of constraints and other manual levers, a developer can exercise significant manual control over this process. This allows for the creation of the highest-performance designs, as well as for running experiments where the default place-and-route process must be modified, such as adding constraints to disallow the usage of soft-logic resources in a particular area [75].

For modern FPGA RTL programming with an extensive library of hard blocks, a robust set of RTL macros is required. These macros offer a relatively high-level interface to the hard blocks, and are often instantiated through GUI-based configuration tools, owing to the deep customization available. Often, macros also have direct support from the FPGA synthesis tool to aid in proper elaboration. Macros and premade FPGA IP can also encompass multiple blocks at once in larger parameterized aggregate structures, hiding the complexity of the internal block based design behind a friendly RTL interface.

High-Level Synthesis. To accelerate the development of FPGA designs, significant work has been done in the field of High-Level Synthesis (HLS), typically described as an "algorithm to hardware" flow, where the designs are written in a more typical language such as C/C++, and then converted to RTL and scheduled onto an FPGA fabric by an HLS compiler [76]. Unlike RTL which is strictly timed and laid out in exhaustive detail, HLS allows the writing of high-level code with high-level specification for performance engineering variables such as the degree of pipelining of a module. Therefore, it can also be seen as a convenient way to "write" good RTL, with high-level directives offloading the complexity of pipelining and scheduling the code to the HLS compiler [77, 78].

The high-level nature of HLS also brings improvements to full-system programmability and dynamic re-programmability. In the Intel oneAPI SYCL for FPGA environment, a domain-specific eDSL for writing heterogeneous acceleration code, both the host code and the device code are written in the same language, allowing both the developer and the compiler to reason about structures across the gap. This feature greatly simplifies the development of libraries for the interaction of FPGAs with not just the host but also other devices, allowing the rapid development of heterogeneous applications [79].

High-Performance HLS Kernels. While HLS is frequently described as above, it is important to remember that a straightforward C/C++ input will not typically result in a good HLS design. This is because, the C/C++ input is used to generate a logical implementation which is then scheduled onto FPGA hardware and RTL code generated out of it. If the C/C++ is written arbitrarily, the compiler will typically not be able to find an optimized way to schedule the HLS code. Therefore, writing high-performance HLS kernels requires a structural way of thinking in which

the structure of a high-performance design is first envisioned, and then expressed in appropriately-written C/C++. This results in frequently strange-looking C/C++ compared to how a software developer would write it, which is simply the result of the FPGA's particualr structure and parameterization. However, the principles are the same as with writing high performance software, in that it is important to expose parallelism, manage dependencies, and ensure correct data movement.

2.3 FPGA IO and Data Movement

2.3.1 High-Speed Data Transfer

Long Distance Serial Links. When the devices are physically separated on a PCB or across a network, high speed serial links carrying very high frequency signals (28GHz+ for FPGA/switches) over differential pairs of wires are used. The two main classes of signaling are **NRZ (non-return-to-zero)**, which uses two voltage levels to represent a 1 bit values (0 and 1); and **PAM4 (pulse amplitude modulation 4-level)**, which uses four voltage levels to represent a 2 bit value (00 to 11) [31].

PAM4 doubles the data transmission rate at the same frequency, but requires more complex analog frontends and is harder to route because of the necessity of needing to distinguish between four voltage levels. It also uses hardware forward error correction to handle the much higher bit error rate. Despite these drawbacks, PAM4 is the signaling standard used both on the FPGAs and the switching ASICs because it cuts the signaling frequency in half compared to NRZ, avoiding costly losses.

In modern deployments, vendors have achieved very high density for serial links, on the scale of what is required for the SOAP project in its largest 1024 element configuration (hundreds of 100G / 200G PAM4 links per array). This is evidenced by

datacenter switches supporting 64x800G configurations, requiring 512x100G serial links inside a single switch, showing that this data transmission density is possible inside a single unit. Examples of such high-density switches include the NVIDIA Spectrum SN5600 [80].

This enormous switching capacity is enabled by very high density switching ASICs that power the above switches. An industry example is the Broadcom Tomahawk series, where the Tomahawk 5 can switch up to 51.2T and the Tomahawk 6 can handle up to 102.4T [81], well in excess of what is required for the SOAP project. However, switching ASICs will not run arbitrary serial protocols and instead require a protocol such as Ethernet. Therefore, for FPGAs to talk to them, protocol hard blocks are required, which are already available in modern FPGAs.

Short Distance Parallel Links. When devices are physically close to each other, such as within the same package, a wide connection made of a large number of wires is used, with each wire being relatively low bandwidth and signaling frequency. For example, the interface of a HBM4 memory stack consists of 2048 wires at 8 Gbps bandwidth of NRZ signaling per wire through a silicon interposer block, resulting in a cumulative bandwidth of 2 TiB/s [82]. Such interfaces require dedicated hardware support (such as the DDR/GDDR interface on most devices, or the HBM interface on accelerators) and require that all the components be packaged together at a dedicated packaging facility. They are able to transfer data much more cheaply than a high speed serial link

2.3.2 Modern FPGA IO

FPGAs can have both high speed serial and parallel interfaces as shown in Figure 2.2. For wide parallel interfaces, for example, Altera Agilex M-series FPGAs support

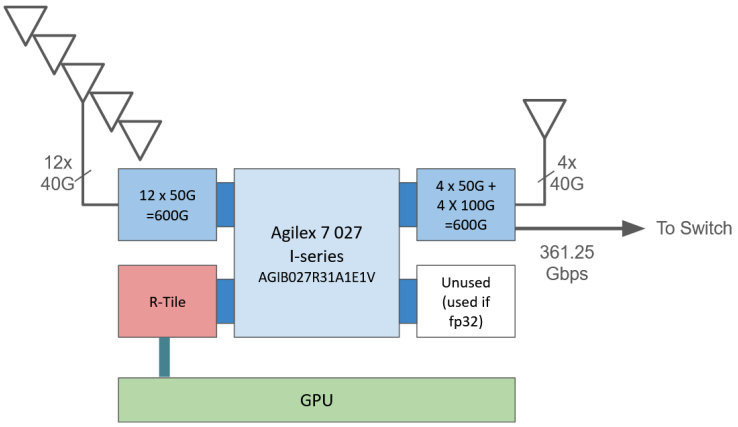


Figure 2.2: FPGA configuration showing diverse IO.

HBM, while other Agilex FPGAs support DDR memory instead. For high speed serial, the modular construction of Agilex FPGAs allows adding in a transceiver block supporting a range of capabilities, along with (bypassable) hard blocks for Ethernet and PCIe. These hard IPs can be bypassed in PMA Direct mode which allows per-lane low-level access to the Physical Medium Attachment (PMA, the analog/mixed signal transceiver) layer but with essential amenities such as hardware Forward Error Correction enabled, allowing for custom protocols. The PMA/FEC Direct PHY IP breaks out serial lanes as parallel interfaces at a data rate supported by the FPGA fabric. The F-tiles contain two types of transceiver lanes: general purpose FGT lanes at up to 58G PAM4 and special FHT lanes at up to 116G PAM4. Each lane is duplex [73].

Device	**Max F-Tiles**	**FGT(58G) + FHT(116G)**	**Total**
AGF 012 / 014	2	24 + 8	2.3 Tbps
AGF 022 / 027	4	48 + 8	3.7 Tbps
AGI 022 / 027	4	48 + 8	3.7 Tbps
AGI 035 / 040	6	72 + 24	6.9 Tbps

Table 2.2: Agilex F-Tile transceiver specifications.

However, not all lanes are always bonded out due to thermal consideration, package and pinout restrictions, or enabled transceiver features in different market segments. Additionally, the size of the fabric determines the available shoreline for IO interfaces and therefore the number of peripheral tiles that can be connected. For example, even though Agilex F-series devices can have up to two F-tiles, the FHT lanes are not bonded out, limiting the maximum spec to 58G PAM4. Agilex I-series devices do bond out FHT lanes, with the largest 040 fabric spec in a R39A package hosting up to 6 F-tiles with all 72 FGT and 24 FHT lanes bonded out for a total of ∼6.9 Tbps of maximum IO (note that each lane is duplex and the bandwidth provided is the one-way number).

2.4 Prior Work in Inline FPGA Computing

The good fit of FPGAs towards streaming computation has been researched extensively, with significant prior literature on streaming and bump-in-the-wire accelerator deployments. This section will survey the landscape of existing work and differentiate the data transport computing problem from them. From the point of view of this work, the prior literature can be categorized into three broad categories: (1) datacenter bump-in-the-wire infrastructure deployment, (2) programmable inline packet processing, and (3) inline services near storage or other data appliances.

2.4.1 Datacenter FPGA Deployment

This class of work stems from large hyperscaler deployments of FPGAs as infrastructure and bespoke application accelerators. The archetypical work in this field is Catapult [83], which first described a deployement of FPGAs in Microsoft datacenters for accelerating Bing search rankings. This first version of Catapult is not a

bump-in-the-wire architecture, rather it is an integrated FPGA alongside the server with its own networking, taking part in a large-scale reconfigurable fabric. This makes the use case different from data transport compute, and more geared towards general-purpose acceleration. However, FPGAs were chosen for this project due to their combination of flexibility and streaming acceleration capability, substantiating the data transport computing claim of the value of flexibility.

The followup the Configurable Cloud architecture [16], is a true bump-in-the-wire design with the FPGA physically placed between the NIC and the switch, which requires every packet to traverse it. This version is presented as a common network plane for software-defined networking, in addition to an application acceleration fabric. Unlike Catapult, this architecture allows for the inline compute required for data transport computing. However, it does not specifically support the computational archetype (short bandwidth-heavy pipelines) and integrates the FPGA quite distantly from the system as well. Moreover, the FPGA remains an off-the-shelf part, with no special effort to optimize it for the given computation. Instead, as in Catapult, the focus of the FPGA here is on flexible acceleration of relatively infrastructure workloads such as network planes.

A third work from the same lineage, AccelNet [17], moves further along the infrastructure computing specialization to offloading the networking stack onto the FPGA. This is not a computing workload as expected by data transport copmuting, but it does underscore FPGA flexibility yet again. In summary, FPGA usage in this domain is majorly about flexible spatial acceleration, with integration modality playing an important role as well (bump-in-the-wire). However, the computations done here do not fall into the data transport computing regime, leaving a gap to be filled.

2.4.2 Programmable Inline Packet Processing

This class of use cases is geared towards creating programmable network functions that adhere to very particular abstractions, enabling high performance streaming implementations on FPGA. An important work here is ClickNP [84], which proposes a streaming abstraction for FPGA-based network packet processing. This abstraction is quite amenable in spirit to the requirements of data transport computing, as it is geared towards streaming pipelines. However, ClickNP is focused exclusively on networking workloads and does not include any FPGA-side fabric or architectural optmizations to support its streaming worldview. It also does not have a particular focus on efficiency other than through creating FPGA implementations. Data transport computing as presented in this work is less focused on abstractions and more on the architectural effects of processing as efficiently as possible within the domain. Other packet processing work which have more constrained but specialized processing interfaces like FlowBlaze [85] and hXDP [86] do have a focus on data bandwidth, pipelines, and efficiency, but fall short in that they attempt to derive FPGA designs from a high-level network-centric abstraction, thus losing generality compared to the data transport computing problem, which applies to all computational units. In summary, in this domain there is a focus on efficient processing of high data bandwidths, however the work is focused on network-centric abstractions and not on a general treatment based on the architecture of the FPGA.

2.4.3 Inline Data-Centric Services

This category covers work on inline processing in the database and analytics domain. It is particularly relevant as this naturally lends itself to the processing of high data bandwidth with the intention of filtering out data for energy and computational

reasons. In [19], the authors argue for very similar ideas, by putting streaming aggregation operators on FPGAs. However, as with the other work imagined so far, the FPGA fabric has been held constant, and no effort is made to solve overheads such as soft logic and routing through hard block specialization which is a core contribution of this dissertation Another example of this class of computation is Ibex [11], which demonstrated a streaming bandwidth filter by pushing the operator down into the FPGA, as well as doing SQL offload along the storage path. With the same critique as above in terms of fabric and optimization, this is still a very relevant prior work and a significant inspiration. Other work in this domain like [65], which proposes an associative lookup kernels, also forms early foundations for the work in this thesis, directly inspiring one of the application domains.

2.4.4 The Gap

In every system above, the FPGA has been treated as a fixed constant, while the application is adapted to it. In this dissertation, the fabric will be changed to reflect the domain, significantly changing the design tradeoff of the FPGA in a manner more similar to the specialization of AI FPGAs [72]. There is also no single work with a focus on a computational archetype that informs the hardware, and a particular scenario to use it in (other than network domain). Additionally, the examined work targets networking-grade bandwidths, not a multi-terabit shoreline. For these reasons, the work presented in this dissertation novel and significantly different from the prior art.

Chapter 3

Data Transport-Centric Design Study for a Digital Beamformer

3.1 Introduction

This chapter presents a design effort for an adaptive digital beamforming system, following the parameters of the DARPA-SOAP challenge. This is an application which has a large amount of data transport in the form of very high bandwidth serial IO that is difficult for modern accelerators to interface with, and a computational pipeline with stages with both low and high reuse and different IO patterns, making it an ideal testbed for data transport computing as presented in Chapter 1. An summary of the problem domain will be presented, however, the purpose of this study is not to go into beamforming algorithms or their tuning, and consequently it will focus on addressing feasibility and computation challenges of the SOAP problem, and not the algorithmic challenge. The key problems solved involve both a computation efficiency problem, solved by using FPGAs for data transport

compute, as well as an IO interface feasibility problem in which the memory-like data modality of the GPU prevents it from directly acting on the input data and instead requires costly buffering stages and proprietary conversion hardware.

In doing so, this study will present a case for adding in FPGA inline acceleration along with a typical GPU design, performing filtering and beamforming tasks that would not be ideal on the GPU. As no single device can handle the required bandwidths, a distributed network of processing elements will be proposed which together span the full compute, adding even more data movement to the system, pushing more computations into the data transport computing domain as data flows multiple times across each processing element in each computational stage. The flexible capabilities of FPGAs will be central to this effort, as data input, data exchange, streaming computation, and interface to GPUs will all be handled on the FPGA simultaneously in multiple pipeline stages of the beamforming algorithm at once. Additionally, an argument will be made for deep, application-specific integration of FPGA and GPU architectures inside each processing element that does not disturb the semantics of either device, allowing off-the-shelf implementations. FPGAs in this sense have become computational "glue", but for a very specific bandwidth-centric domain of computation. Together, this study acts as an example of the benefits of FPGA inline acceleration, as well as a baseline for further investigation. This study will treat the IO side of the computation as the core, first-class problem. The results will be used to substantiate the first claim of the thesis, that FPGAs today already solve the data transport computing problem effectively, achieving higher efficiency than a traditional architecture such as a GPU. The subsequent chapters will also use the kernels from this chapter as comparison points.

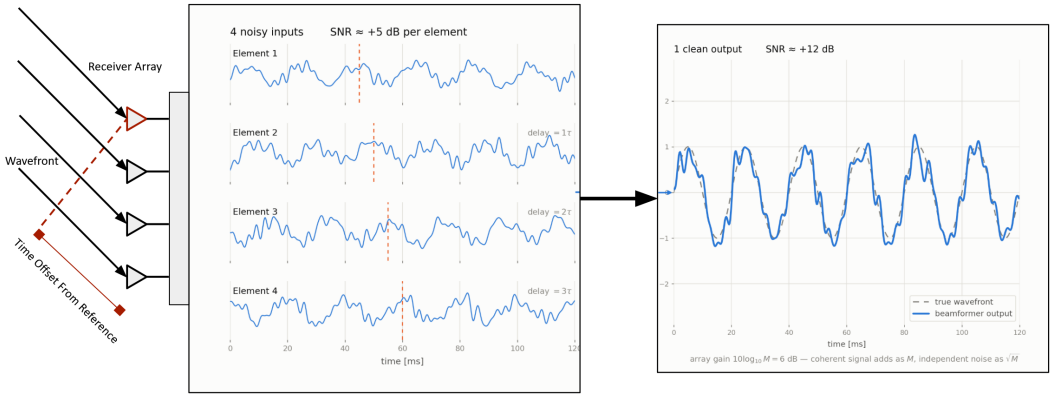


Figure 3.1: A simple 4-element example of beamforming showing the recovered sinusoid.

3.2 Motivation and Problem

3.2.1 Adaptive Digital Beamforming

Beamforming. In signal receivers, such as microphones, radar elements, etc., a key problem is isolating a particular signal coming from a particular direction. Consider as in Figure 3.1 a receiver located on a 1D axis with a collection of signal sources distributed around it. The overall signal at any time in a given receiver is the sum of the signals transmitted, attenuated by propagation losses and directionality of the transmitters and receivers. Since the overall signal is a direct sum, it is difficult for a receiver to tell which direction a signal is coming from and to isolate a particular signal to the exclusion of the others. A commonly-used method of mitigating this is phased arrays, which is an array of receivers physically separated in space at known locations. As in the leftmost (receiving) side of Figure 3.1, the known geometry of the array as well as the known incident direction allows one to calculate the time delay that the same wavefront will be received at any element compared to a reference. The signals from each element can then be timeshifted to line up exactly

as if coming from that direction, constructively interfering for any signal from that direction and destructively interfering for others. This is called beamforming and allows the receiver array to be electronically "pointed" along any direction, isolating that particular signal [87].

Adaptive Beamforming. The key issue is that the presented beamformer acts independently of the environment around it i.e. that it is *non-adaptive*. This stems from the fact that the only input to the algorithm is the scanning direction and the array geometry, and all it has to work with is the mechanism of constructive interference. This works well for simple clean environments, but does not work well when there are a lot of other signals in the area. At an intuitive level, the non-adaptive beamformer is unable to take into account the structure and power of other signals when asked to isolate a particular signal from a particular direction.

To solve this, a foundational class of *adaptive* beamformers is the *Minimum-Variance Distortionless Response* (MVDR) beamformer [61], a widely used algorithm either directly or with some application specific tunings and changes. Intuitively, the MVDR beamformer captures a mathematical description of the signal environment during a particular time range using the *covariance* of the signal samples of the array in that range. The inverse of this covariance is then used to further clean up the output of the beamformer, ensuring that only the desired signal from the desired direction is present in the output.

MVDR operates by calculating a weight per element, which is then used to combine that element's signals with the others in a weighted sum, expressible as an inner product. Mathematically, the MVDR weights are calculated using a simple expression

$$w = \frac{R^{-1}s}{s^H R^{-1} s} \tag{3.1}$$

where R is the covariance of the input samples X during a given time range, calculated as $R = XX^H$ (outer product, often undersampled and also requiring normalization). s is the geometry-dependent steering vector of the array that captures the delay relationship between the elements of the array and is a constant once calculated. In practice, this is seen instead as a linear system and solved with a Cholesky decomposition and two triangular solves. Once the weights are available, the final signal can be obtained by a simple set of dot products of the signal matrix with the weights,

$$f = w^H X \tag{3.2}$$

An important caveat is then that the given algorithm is only applicable for *narrowband* signals (signals where the frequency range is small). This is due to the mechanism of multiplying signals by complex weights (phase shifts), which is only valid for a narrow band around the given frequency at which the phase shift is calculated from the timeshift. Therefore, for a wideband signal, it must first be split into a number of narrow frequency bands [20], each band being then beamformed separately before recombining them at the output, as derived from the suggested reference architecture for the scenarios.

3.2.2 The Scenario

The following problem setup follows the particulars of one scenario in the DARPA Scalable On-Array Processing (SOAP) challenge, which consisted of scenarios in

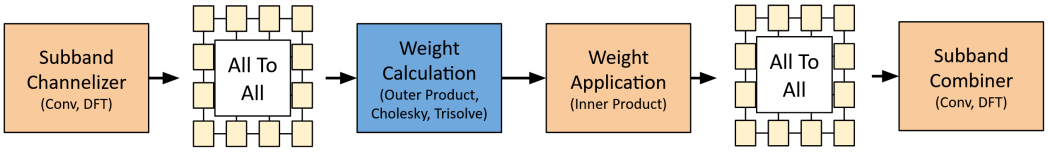


Figure 3.2: 6-stage wideband beamforming pipeline used for this study.

which beamforming needed to be applied. As this dissertation is about computational treatment and not radar DSP, only the relevant details have been provided. In the challenge, a variable but large number of radar elements (128–1024) are recording at a very large quadrature sample rate (1.25 GHz), which at a complex int16 input data type results in an input bandwidth of 40 gigabits per second (40G). **This creates the first challenge, which is simply ingesting and working with a very large amount of total input IO (40G $\times$ 1024 = 40.96T).**

The signals of interest are linear frequency modulated (LFM) pulses with a bandwidth of 520 MHz (approximately half of the sampling rate), with each pulse being 8 microseconds long in an interval of 80 microseconds. These 128–1024 40G inputs must therefore be ingested by the system and beamformed into a single output of 20 beams. The goal was to perform adaptive digital beamforming in this setup at a high energy efficiency (ops/W) while maintaining performance. **This creates the second challenge, which is performing the computations at high efficiency.** The comparison will be done between a GPU-only system under the very optimistic assumption that IO challenges do not exist, and the more realistic FPGA+GPU system. This is directly compatible with the first claim of the thesis, that FPGAs can solve the data transport computing problem effectively already, and can therefore be used to demonstrate it.

A Python-based framework was created using NumPy [88] to perform signal processing experiments and derive the algorithm and parameters required to success-

fully recover the scenario signals. The algorithm is based on the provided reference architecture. For this study, only the baseline MVDR-based wideband beamformer will be considered. The algorithm consists of six stages shown in Figure 3.2. The first stage splits each input (per-element signal stream) into 128 frequency bands, with the middle 64 being used for beamforming (as it is known that only the central 520MHz contains useful data). Then, the data is rearranged from all frequencies of a given element grouped together into all elements of a given frequency grouped together, which is an all-to-all exchange operation. Per band, MVDR weights are now calculated for 20 output beams according to the equations above. The weights are then used to carry out beamforming per band, reducing all elements of a given band into all beams of a given band. Then, another exchange converts this into groupings of all bands of a given beam, which is then recombined to form the final beams.

3.3 Design Study

3.3.1 Processing Grid

Due to the large scale involved, a distributed grid of processing elements (PEs) is used instead of a single device as shown in Figure 3.3. Each PE is logically connected to the inter-PE network by a high-bandwidth link. As the algorithm is mapped to the grid in a pipelined manner and the PEs are all the same, each PE must provision for each stage of the pipeline. The two all-to-all exchanges can be used to inform the bandwidth of the PE to and from the inter-PE network. From the scenario and algorithm description, it can be seen that the first stage is per-element, the second stage is per-frequency band, while the last stage is per-output beam. Each PE is therefore provisioned by the total number of elements, bands, and beams mapped

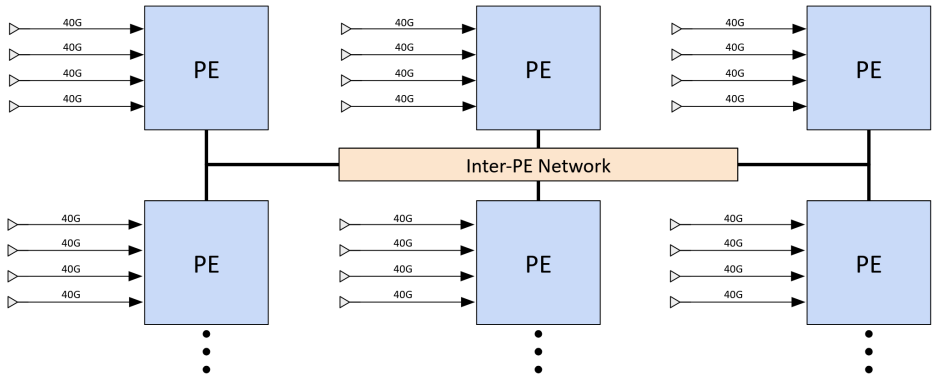


Figure 3.3: Distributed grid of processing elements arranged to solve the beamforming problem.

to it, carried out by a simple greedy mapping program that loops through every configuration and tries to fit as many stages to each PE as possible, given a problem size and PE device configuration.

Going with the restriction on using off-the-shelf parts to not invent new accelerators, the two options are either **GPUs** or **GPU+FPGA hybrids**. The following compares between these two in terms of the IO and interface challenge as well as the computational efficiency challenge, providing evidence to choose one over the other.

3.3.2 IO and Interface Challenges

Ingesting Data from Elements. At the provided 1.25 GHz sampling rate and complex int16 data type, each radar element produces a 40G input stream that must be ingested into the processing system as in Figure 3.4. If this data was available in a centralized high-performance memory system such as the multiple stacks of HBM3e found in a modern GPU or AI accelerator, it could feasibly be read in for computation. For example, even at the largest 1024 case of 40960G or 40.96T or

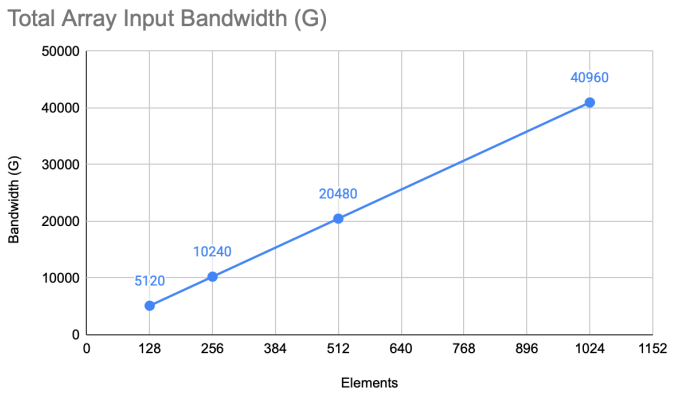


Figure 3.4: Element data ingest bandwidth.

5.12 TB/s, it can be ingested into an NVIDIA B200 GPU with its 8 TB/s of HBM3e bandwidth [89].

However, this is not the case. The data is being streamed in from physically separated discrete elements located across a geometry that can be up to ∼2m on the longer side (3.3cm * 64 elements). As discussed in Chapter 2, the wide parallel interface of HBM, which involves thousands of low-frequency wires over a silicon interposer to reach the reported data rates, is not viable for communication at this distance. Therefore, high speed serial links should be used instead.

If the PEs were made up of just GPUs, as would be the case for a first-cut off-the-shelf design, they would run into problems interfacing with the radar element data input. This is because GPUs either take input from PCIe, as in consumer and workstation GPU cards, or from proprietary inputs such as NVIDIA's NVLink, which is an inter-GPU connection used in AI workloads. A GPU cannot directly interface with a streaming raw serial or Ethernet link as it has no interface to do so.

To interface the data with the GPU at the required rates, the following options are available.

1. **Element sends NVlink to GPU.** This is NVIDIA's proprietary, memory-centric protocol for GPU-to-GPU communication in large clusters, deployed regularly for AI workloads [2, 41]. To talk to an NVlink switch or GPU, the element data stream must be transmitted using a proprietary NVlink PHY and IP, which requires proprietary hardware and licensing. However, while it is technically feasible to do so, the NVlink protocol exposes a memory-centric, atomic operation supporting viewpoint that is not particularly well suited for the "infinitely" streaming element data. Additionally, this will buffer data in a GPU VRAM, from where data will be read into the GPU core. GPU memory hierarchies are optimized for data reuse, which is not that prevalent in this application. Therefore, on top of requiring additional transmission hardware and licensing, this option incurs the extra energy of buffering in a GPU's memory hierarchy for no particular benefit.

2. **Element sends Ethernet to GPU/ASIC.** If the element data stream can be encapsulated in an Ethernet packet, it could be routed through an Ethernet switch into the processing system. While it avoids NVlink licensing costs, and the transmission protocol of low-level Ethernet is quite suitable for streaming data, it still requires that the element has a hardware module to transmit in Ethernet packets, and that the processing system is able to ingest Ethernet data at the given rate. On a GPU system, this requires a NIC to bridge the gap, and suffers from the same issues of extra buffering for no real benefit. Additionally, the ASIC solution requires creating a custom ASIC with both processing capability and an IO shoreline with transceiver and Ethernet hard IP able to handle the required Ethernet bandwidth.

3. **Element sends Raw/Custom Bits, Received by FPGA.** As noted above

in Chapter 2, FPGA transceiver tiles can directly ingest raw bits for use in custom protocols, or use hard IP to decode protocols such as Ethernet. The 40G requirement per physical element also maps to a single 58G PAM4 FGT lane of an Agilex F-tile transceiver with a lot of bandwidth to spare for additional operations such as extra error correction on top of error correction already performed by the PMA (Physical Medium Attachment, the analog frontend) and FEC (Forward Error Correction, the standard line hardware error correction) units. With 12 such lanes per F-tile, a single FPGA can ingest element data in any protocol from 12 elements, with this capability only scaling up as additional tiles are added. The FHT lanes can also be used for this purpose providing additional lane capacity as required increasing the element count to 16 per FPGA (despite the 116G FHT lane capability, the element will only send one 40G stream). This simplifies the element module to also send raw bits (with error correction). Additionally, since the FPGA can operate on the stream directly without needing to buffer in SRAM, it removes the energy of a DRAM hop, which can be significant compared to compute energy. The FPGA can also communicate with a GPU over PCIe or a more custom protocol while avoiding expensive custom hardware.

From these considerations, avoiding custom hardware requires using an FPGA to interface with the input streams for a low energy implementation, communicating with a GPU as required over PCIe or a more proprietary interface. This sets the only feasible composition of the PE to be the FPGA+GPU hybrid.

Element Data Distribution and Locality. Beamforming as an operation finally requires data from all elements to coalesce into a single place for a given band, from which a single resultant stream per beam can be emitted. These are the two

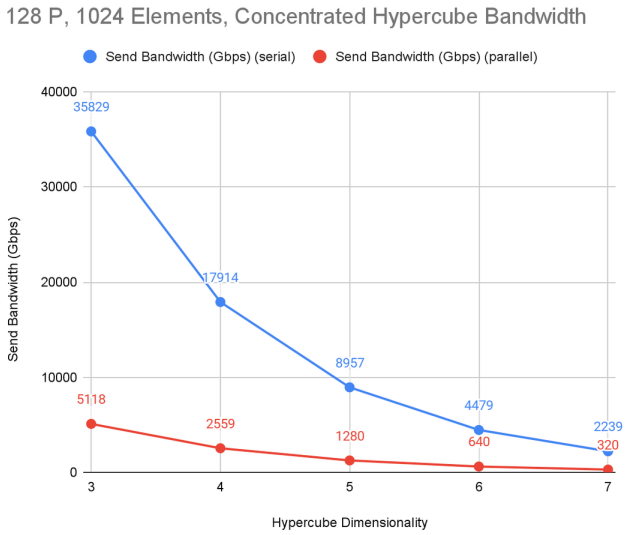


Figure 3.5: Hypercube bandwidth depending on cube order, showing the very large bandwidth that would have to be switched by each node.

data exchange operations in the algorithm above. If multiple processors are used, each that handles one or more of the partitions outlined above, a sufficient data distribution network is required. Assuming fully disjoint partitioning into subbands, this all-exchange will require sending and receiving data at the data ingest rate, and depending on network topology, this multiplier may be many times higher. For example, if a hypercube network was used, each endpoint (FPGA) would have to switch an enormous amount of data as shown in Figure 3.5. This problem was ultimately solved by specifying a relatively recent technology, that being datacenter switching ASICs, which offer the required bandwidth in a single package with 100G and 200G transceivers, which can be interfaced with using the Ethernet hard blocks on the FPGA side. It just requires ensuring that the switched network bandwidths are within the 51.2 Tbps+ capacities of modern switching ASICs [81].

The last detail that must be considered is the interfaces between all the devices. It is assumed that the radar elements operate a serial link to the FPGA, which

is consumed by 50G PAM4 serdes. The FPGA-to-switching ASIC connection also operates over PCB copper traces, with the total required density already handled by the higher end of commercially available PCB or flyover cable technology used by AI datacenter spine switches in their highest configurations, such as 64x800G. However, the FPGA-GPU bridge is a significant undertaking. Currently, the only possible technology is PCIe, which is a significant issue for the manner of low latency streaming connection that is desired. However, as that is what is available at the moment, a version of a system with PCIe is detailed in the implementation section. If a more suitable interface can be conjectured, such as a lower level interface directly using the High-Bandwidth Memory interface on the cache, significant system integration benefits are possible. A version of the system with this configuration in mind is also presented.

With all the above considerations, the only feasible option is to use the FPGA+GPU hybrid. From the perspective of data transport computing, this is an interface impedance mismatch where a streaming input must be ingested by a system that is not built for it, requiiring expensive and proprietary conversion, lowering system efficiency. The modern FPGA can therefore be seen to be solving this leg of the data transport computing problem through its diverse set of IO interfaces, substantiating half of the thesis claim that FPGAs today already solve the given problem domain.

3.3.3 Kernel Efficiency

High Efficiency Through Minimal Buffering. To achieve the high efficiency at high performance goal, a low-buffering approach is followed here. From Table 1.1 it can be seen that memory, and buffering in general, is a significant source of

power in a system, therefore it is good to avoid trips through DRAM systems. This requires that the PE perform all its processing using only on-chip memory. From Little's Law [90], a large occupancy requires a large amount of buffering, which would almost certainly fall into DRAM territory due to the limited availability of on-chip SRAM buffers in all devices. Therefore, achieving the required throughput at minimal latency is good for efficiency. **Towards that goal, DRAM is not used at all in this design, and all stages only use on-chip buffering.**

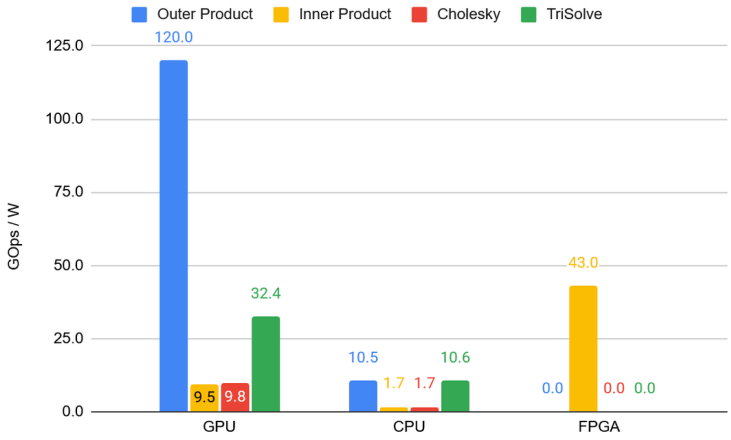


Figure 3.6: Efficiency of beamforming kernels on various devices.

Baseline Kernel Efficiency. As shown in Figure 3.6, the GPU achieves good efficiency on the covariance calculation (the outer product kernels), but quite poor efficiency on the beamforming operation itself (the inner product kernel), measured through repeated executions of CuPy [91] calls. An accounting of the operations of the beamforming pipeline (only the base pipeline without the channelizers), shows that the outer product contributes most of the operations, however, as they are dealt with relatively swiftly, most of the power consumption actually comes from the inner product kernel. This performs poorly, as there is very little data reuse for

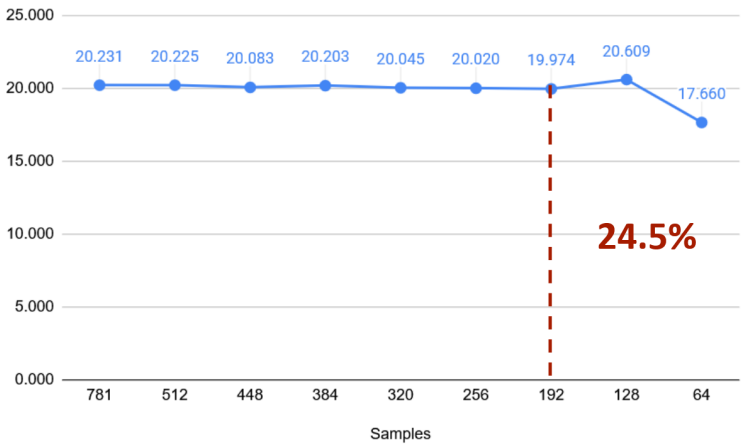


Figure 3.7: Random sampling threshold determined empirically. The brief increase in SNR happens as low-quality signals are lost, increasing the SNR of the ones still visible.

each element sample due to the involved shapes. This shows that **even if the GPU could solve the IO interface problem all by itself, it would still result in an inefficient system due to poor performance in the inner product kernel.** This happens to be a data transport compute kernel as streaming samples are computed against a stationary weight matrix, with a broadcast of 20 per input sample as each needs to be multiply-accumulated with each beam weight.

The FPGA is much more efficient than the GPU at this kernel due to this streaming structure. Constructing an appropriate FPGA inner product kernel implementation and benchmarking it on streaming data, the FPGA achieves more than four times the efficiency of the GPU on this kernel. By including it in the system, the drawbacks of the inner product kernels are significantly reduced.

Bandwidth Reduction. A significant caveat from a naive IO design would be that the full input bandwidth needs to be transferred to the GPU. Although the GPU is now reached over PCIe in design and is interface-compatible, this still requires

transferring the entire 5.12 TB/s over to the GPU. As mentioned in Chapter 1, another common use case for data transport computing is bandwidth reduction, leading to computational reduction in the output. As it happens, algorithmically the GPU does not actually need all the input samples to calculate a good enough estimate of covariance. Instead, a random sampling of the samples can be sent, leading to much the same effect. This was verified using the Python framework and, for the right scenario, measured to limit out at about 24.5% (Figure 3.7) of the input data, by running the scenario through the framework and random sampling until the covariance degraded enough to cause a significant loss of accuracy. Therefore, in the final accounting, the GPU covariance only has to perform its operations on a much smaller input.

Additionally, as only the middle 64 frequency bands out of 128 carry relevant data, due to the 520MHz bandwidth of the pulse out of a total sampling rate of 1.25GHz, the upper 64 subbands (the upper 32 along both positive and negative directions) can be safely discarded as they contain no useful data about the pulse at all. This is another "free" bandwidth reduction that decreases the requirements on the GPU while costing essentially nothing on the FPGA, while on a GPU-only system it would have required ingesting all the data through an expensive interface, only to throw most of it away after some relatively cheap computation. While this has the effects of reducing its dominance in the operations, the major effect is to make the system design more viable by decreasing the amount that the FPGA has to transfer over PCIe and thus system resources. It illustrates a key usage modality of data transport compute as a downstream bandwidth and computation reducer.

3.3.4 System Sizing Model

To size the PE grid and interconnect, a design sweep is done using a simple resource model for both the beamforming problem as well the FPGA and interconnect resources. The following describes the assumptions used in constructing this sizing model.

1. Compute block count is used as a proxy for FPGA resources. However, 100% block usage is not realistic since such a design would be essentially impossible to route and does not count the overhead of infrastructure and management IPs. Therefore, usage was capped at 75% compute blocks, which is still difficult but more achievable, especially with repeated copies of a modular design that make routing easier for the FPGA tool.

2. Element input is done via 50G transceiver lanes, of which there are 12 usable per tile. If required, 100G lanes can also be used, of which there are 4 usable per tile. Through trials in Altera Quartus, it was determined that while each F-tile has about 1200G capacity on paper, in practice using error correction hardware is limited to about 600G per tile, which determines the lane configuration constraints.

3. Each processor inputs CEILING(NumElements/NumProcessors) elements and is responsible for CEILING(NumSubbands/NumProcessors) subbands. Two interconnect options are available: the FPGAs can be connected directly to each other, or centrally through a switching ASIC. Processors will keep the fraction of subband data available to them and send the rest along to the network.

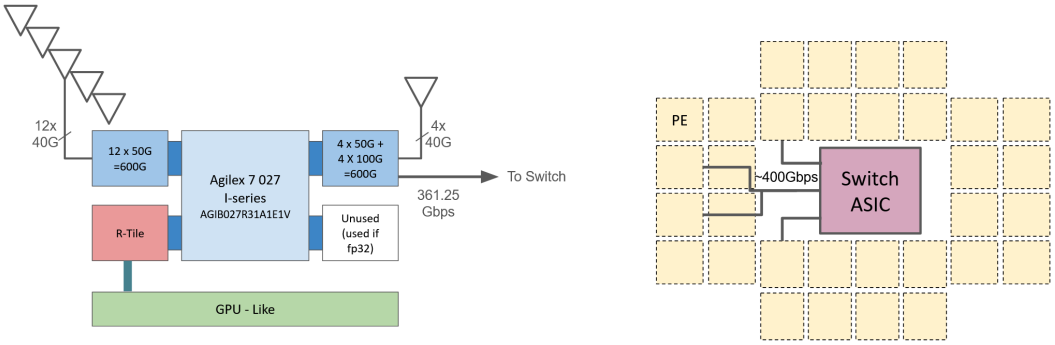


Figure 3.8: FPGA configuration showing diverse IO.

GPU Communication and Buffering. The core problem being solved with the IO interface is that GPU interfaces are not well-suited to FPGA styles. However, communication with the GPU is inevitable and can be done in two ways today: over PCIe, over a shared memory interface (assuming a common RAM). The former is viable now as the FPGA can use its transceiver tiles to implement PCIe, and the latter can, in theory, be implemented via a custom protocol. However, ideally one would want a wide parallel interface to the GPU, which can be proxied by an HBM interface. While this is not present in devices at the moment, it is the ideal interface for such an FPGA and is proposed as a limiting case below.

These constraints are written into and solved by a Python program which loops over all processor counts from 1 to 64 (beams are handled internally). The valid configurations are presented below.

3.4 Results

The final design is a grid of FPGA+GPU hybrid processing elements, connected in the middle by a switching ASIC. The above design space search process fits a given problem size to the array, resulting in the following design points. Figure 3.8

Elements	PE Count	Elements per PE	Subbands per PE	Array Input (Tbps)
128	8	16	8	5.12
256	16	16	4	10.24
512	32	16	2	20.48
1024	64	16	1	40.96

Table 3.1: PE parameters.

	Unidirectional bandwidth (Gbps)			
Elements	Subbands	Beams	Final Out	Total Out
128	280	50.00	120	450.00
256	300	25.00	80	405.00
512	310	12.50	40	362.50
1024	315	6.25	40	361.25

Table 3.2: Link unidirectional bandwidth.

shows an instance of the design, showing an FPGA with its myriad IO configurations and the overall system grid. Parameters of the grid are presented in Table 3.1 and Table 3.2.

3.4.1 FPGA Configuration

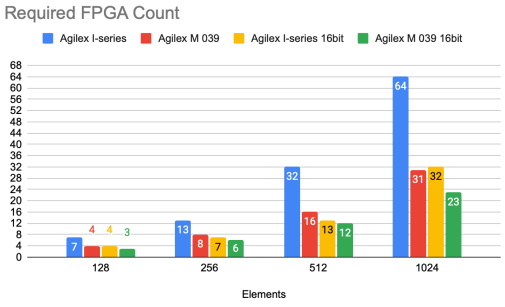


Figure 3.9: Required FPGA count from design sweeps.

Elements	Bandwidth (Tbps)
128	3.60
256	6.48
512	11.60
1024	23.12

Table 3.3: Total bandwidth handled by the switching ASIC.

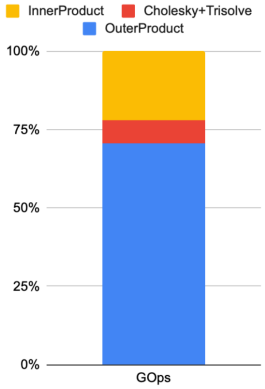
Elements	**Agilex I-series (Gbps)**	**Agilex M-series (Gbps)**
128	997	1510
256	954	1445
512	723	1405
1024	721	1431

Table 3.4: Required out bandwidth to switching ASIC.

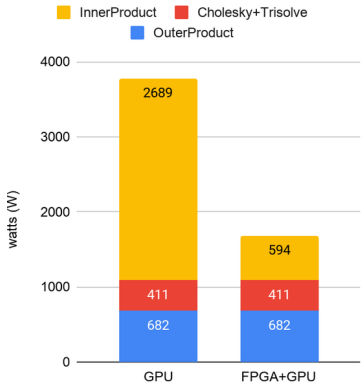
FPGA Choice. Here, two FPGA families have been used corresponding to two choices of instantiation. If only PCIe interface to GPU is allowed as is true today, the Agilex I-series is the stand-in for that system, with a large general purpose transceiver count. For a theoretical "speed of light" system where an HBM-like wide parallel interface to the GPU is possible, the Agilex M-series is the stand-in for that system, as its inbuilt HBM interface allows one to treat is as a proxy for that. The advantages of a wide high-bandwidth link to the GPU are readily apparent, as the IO that could have gone to more interconnect bandwidth is instead redirected to the GPU. This results in the M-series proxy achieving a higher PE density as in Figure 3.9, driving the total switch bandwidth in Table 3.3 with a higher per-FPGA bandwidth as in Table 3.4.

3.4.2 Compute

GPU-Only Compared to GPU+FPGA Approach. For a given 128-element antenna configuration, it may seem reasonable to use a GPU for the whole pipeline,



(a) Total ops in baseline beamforming pipeline.



(b) Adding an FPGA reduces system energy by 55.3% due to prevalence of inner product operations.

Figure 3.10: Base beamforming pipeline ops and energy, showing efficiency enhanced by adding an FPGA to perform the operations unsuitable on GPU.

given that in a breakdown of the operations, the large majority of them are from the two matrix multiplications. However, while the GPU shows good efficiency for the outer product, it is much worse at the inner product (below numbers are DRAM-to-DRAM) as seen in Figure 3.10. This is due to the low data reuse possible in the inner product relative to the large overhead of data streaming on the GPU, where an incoming streaming element has to cross the IO stack, memory crossbar, various caches, and finally reach the registers only to be used only once per output beam. In contrast, an FPGA implementation of the same simply streams the incoming data to an array of hard multiplier and adder blocks, one per beam, reaching full utilization of the array and therefore much higher efficiency from simply not using the rest of the fabric. Here, adding the FPGA reduced the total power by 55.3%.

Full-System Compute. Comparing the FPGA power to the equivalent a GPU would have used shows a constant reduction with increasing problem size as the data transport compute kernels scale linearly with data transport volume, unlike

Kernel	TOps/s	GOps/W	Total W	Device
Analysis Conv, 2 pt	2.24	61.44	36.45	Agilex 7
Analysis FFT	5.60	20.17	277.55	Agilex 7
Covariance	20.06	120.00	167.15	RTX 4090
Cholesky	2.24	9.76	229.20	RTX 4090
TriSolve	2.10	32.44	64.65	RTX 4090
Inner Product	12.77	34.17	373.75	Agilex 7
Synthesis IFFT	0.87	20.17	43.37	Agilex 7
Synthesis Conv, 2 pt	0.35	61.44	5.69	Agilex 7
Total	**46.22**		**1197.80**	
Average		**38.59**		

Table 3.5: Per-kernel throughput, efficiency, and power by device.

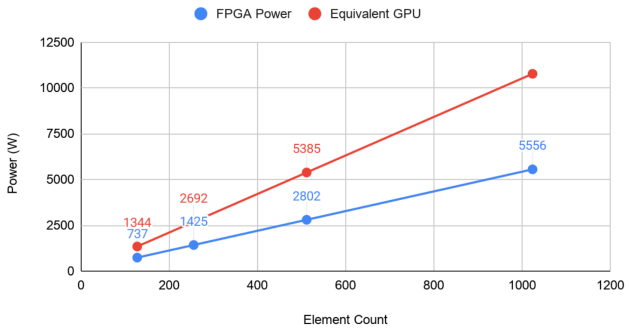


Figure 3.11: FPGA transport kernel power compared to equivalent GPU implementation, showing FPGA architectural benefits.

the weight estimation kernels which scale superlinearly. This is shown using the full-system accounting in Table 3.5 and visualized in Figure 3.11.

As the problem scales, the FPGA power increases linearly while the GPU power increases superlinearly. **This is a property of the algorithm and not a data movement computation problem. As seen above, the power for movement kernels has been reduced and handled. Algorithmic optimization can now focus on the GPU side, as was the case in the original SOAP**

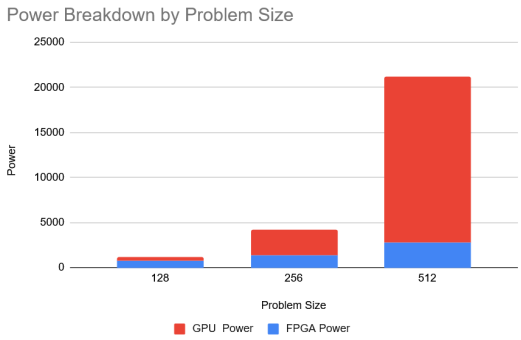


Figure 3.12: Fraction of FPGA vs GPU power as problem size increases.

Figure 3.13: Best-case problem fit, showing current bottlenecks.

challenge. This is orthogonal and not explored here. It is demonstrated in Figure 3.12.

FPGA Provisioning. An important question answered during the design sweeps was how well-tuned a current FPGA is for this kind of workload. Using the absolutely best-case fabric with the best case assumptions, in which the interface problem is solved by connecting to the GPU over a custom wide parallel link (an unreasonable assumption for a device today, but useful as a limit), the following was observed.

From Figure 3.13, it can be seen that the FPGA is bottlenecked by its transceiver capacity, while strongly overprovisioned for compute (only 24% of compute blocks are instantiated). Additionally, it does have enough bandwidth through its HBM interface for GPU communication, which, once again, must be emphasized

that is a limit and not something the device can do today.

3.5 Conclusion

In conclusion, it has been shown that FPGAs fulfill both the requirements for solving a data transport compute problem, in that they provided a more efficient implementation of the relevant kernels, as well as a good system integration option that is both feasible and enables efficiency. This demonstrates the first claim of the thesis. A key facet that is brought out in this chapter is the importance of judging IO, and the issues that occur when this is not taken into account. It was also shown that the FPGA today is transceiver-limited for this class of problem, while being over-provisioned on compute, implying that an ideal FPGA instantiates more IO and moves data to a smaller amount of compute faster. Another key finding was the importance of matching interfaces, such as the radar-to-FPGA interface (a streaming interface which matched on both sides), or the FPGA-to-GPU interface, which was a poor match both physically and logically. This will be expanded upon in Chapter 6 with a deeper discussion of system integration.

Chapter 4

The ASIC Gap, Quantified

In this chapter, the various architectural tradeoffs of the FPGA are evaluated against an ASIC baseline to quantify the origin of the efficiency gap with the ASIC. For this, an ASIC baseline model is first constructed from surveys of relevant literature as well as performance and power results from RTL synthesis and place-and-route based on the open-source OpenROAD and ASAP7 toolchain [92, 93]. A like-for-like comparison is then done against this model to identify areas where the FPGA is losing out in performance and power, with the working hypothesis being the generalized routing and LUT structure of the device. This point can already be seen in the literature [27] with the claim that hard blocks are already ASIC-like, so the difference must be in the soft logic and routing. However, this is not an accurate or detailed enough look for the purposes of this dissertation and therefore bears additional quantification. Following this, the next chapter then constructs an optimized FPGA for the data transport computing role. For the thesis, this chapter plays an important part in anchoring the FPGA efficiency claim to a more objective baseline, as well as providing information for the optimization effort for the second

half of the thesis.

4.1 Motivation

In the previous chapter, it was shown that FPGAs can add significant efficiency to a baseline GPU-only system under the principle of using each device in its own best interest, namely, using the GPU for dense matrix operations with significant data reuse and the FPGA for streaming inline compute with little data reuse. This study instead looks to characterize the cost of the architectural tradeoffs and choices made by today's FPGAs, with a view towards eventually changing some of those choices in the next study. In this study, the following questions are asked and answered: **How does the FPGA compare to a generalized ASIC baseline under data transport workloads? Further, what exactly in the FPGA's architecture (and microarchitecture) causes these inefficiencies and can they be characterized for this transport-dominated workload?** These results will then be used in the next study to explore an FPGA that is optimized for these data transport streaming use cases.

4.1.1 Scope and Limitations

This study seeks to evaluate the modern FPGA architecture against a constructed ASIC baseline for a small set of streaming kernels drawn from the previous chapter, namely filters, DFT-128, and inner product. This is due to their relatively simple computational structure which aids the comparison being made here, as well as compute-centric data transport compute kernels placing higher arithmetic demand on the fabric, making distinguishing them from other regular compute kernels more difficult (string and hashtable based kernels like aggregation and the multi-string

pattern matcher are trivially differentiated). The ASIC models will be constructed according to the methodology below, and microbenchmarked for a detailed breakdown of power consumption in various components such as arithmetic engines, data movement structures, SRAM buffers, etc. Similarly, using FPGA EDA tools, a similar breakdown is constructed of the FPGA kernels on a commercial FPGA device today, with care taken to ensure a similar comparison across devices, namely the comparison being done iso-throughput, with similar inputs, and with a similar dataflow to the extent possible.

Using these breakdowns, a per-kernel comparison will be done of the performance and power characteristics of structures with similar roles (computation, data movement etc.) on both sides. From this, it is expected to reveal the relative shortcomings of the FPGA in the components of its tasks compared to a theoretical ASIC peak.

However, the lack of a direct ASIC comparison comes with a few significant caveats. While the extensive published literature on ASIC accelerators can be surveyed for comparison points and correctness anchors, the numbers so obtained are, at best, rough approximations for the purpose of this study. This is due to the fact that ASIC designs are very specialized for the given algorithm and operating conditions, and generalizations thereof are bound to be inaccurate, as there is no single ASIC that perfectly replicates the kernels being evaluated under the conditions being evaluated. Moreover, the open-source-based methodology of creating the baseline through the assembly of results of simple kernels, buffers etc. is naturally inaccurate compared to the extensive and detailed treatment given to ASICs in production. However, the goal of the ASIC comparison is to create a rough approximation of an ideal, something that can be used both as a comparison point

and as a goal for further optimization.

Several factors in the methodology systematically or practically underestimate the ASIC's true efficiency, making the gap to the FPGA wider than shown. This does not change the general argument or the observed trends; however, it does change the quality of the results. Firstly, the open-source Yosys-based synthesis toolchain [94] is documented to materially predict a higher power than the actual [95]. This has been somewhat accounted for by giving the toolchain a very simple operator one by one and composing the final result manually, so it does not actually have to handle a large design at once. However, the trend still exists. Additionally, the precise numbers are strongly dependent on the exact implementation of the operators used. The operators considered here were constructed manually, but do use the `-booth` option to force good multiplier construction from Yosys, and are pipelined to reach the correct operating frequency. However, even if all this was perfect, they are still designs synthesized out of standard cells that are less efficient than commercial macros. Finally, the ASAP7 PDK used here is predictive at best and not a real replacement from a commercial 7nm PDK. However, this is defended against in two ways, namely, firstly the claim that FPGA hard blocks are ASIC-tier [27], as well as the very simple and individual nature of each operator, which forestalls most of the larger-scale optimization that a commercial flow would use.

4.2 Methodology

The following describes the general high-level construction and methodology of the ASIC and FPGA sides in the study, as well as the comparisons constructed. This chapter is concerned with modeling the ASIC baseline and the evaluation of a commercial FPGA against the baseline in a like-for-like comparison. As there is no

exact ASIC that perfectly replicates the functionality the FPGA side performs, the scope of the comparison has been kept relatively small to allow for a fair like-to-like analysis. The salient assumptions are as follows.

Component	**Basis**
FPGA — Agilex 7, Quartus PA, Quartus PTC	
Logic and signal routing, DSP, M20K, Clock	Measured
Static power and shell blocks	Measured
ASIC Baseline — ASAP7 through OpenROAD	
Datapath operators, steering and register blocks	Measured
Clock trees and leakage	Measured
SRAM arrays	Measured and literature
Process Node	Literature and modelled
Wire Energy	Literature
Design Composition, Floorplan	Modelled

Table 4.1: Components characterized on each side of the FPGA and ASIC comparison.

Kernel-Systems Only. Only kernel systems are compared between device options. This is deliberately done to isolate the comparison from the details of platform differences, which can skew the comparison. Additionally, FPGA and ASIC share similar cost for external structures, such as transceivers, causing them to be roughly equal. The modeled components are listed in Table 4.1.

Iso-throughput and iso-clock comparison. The comparison is done iso-throughput, and at iso-clock and 2x clock. This is because, in the pipeline from the previous chapter, the throughput is governed by how many samples are handled within a cycle. With a sampling rate of 1.25 GSa/s, this corresponds to handling 1 sample / cycle at 1.25 GHz or 2 samples / cycle at 625 MHz. The FPGA implementation does the latter, as the former is not a reasonable FMax to hit with current-generation

FPGA devices. Therefore, the 625 MHz ASIC serves as a direct point of comparison. However, modern FPGA structures can clock reasonably close to or at 1 GHz [71], therefore the 1.25 GHz point is also synthesized as a check; however, its results do not change the conclusions. While it is true that ASIC frequencies can be pushed far beyond this number, this is not in scope for this thesis, and, as presented above, a better-optimized ASIC is simply a stronger baseline and does not change the essence of the comparison.

Toggle Rate Ranges. Both FPGA and ASIC side tools provide for both single-toggle-rate and trace-driven power simulation. This can have a considerable effect on comparisons, as activity factors determined by these tools can have a large effect on power. In this study, a single toggle rate is used for full power analysis. A high toggle rate (50%) represents uncorrelated inputs and is a reasonable worst case for exercising each architectural feature. This can over-represent the power, as this high a toggle rate does not occur that often at the input in practice. A lower toggle rate implies lower overall dynamic power as the circuits have to switch less often. The most significant difference this makes is that each micro-structure is affected differently by different toggle rates, such as wires being very highly affected, while the overall execution of a compute block is less so as it is a complex of some structures that always toggle and some that stay constant (consider a higher bit in a calculation that is not affected much by changes in lower bits). Therefore at lower toggle rates, compute terms grow larger, while at higher toggle rates, data movement terms grow larger. In order to ensure a fair comparison, the models were evaluated at toggle rates of 12.5%, 25%, and 50%, ranging over reasonably correlated to uncorrelated inputs. A more accurate method would have been annotating the netlists with activity from a simulation. This was tried but ultimately failed to

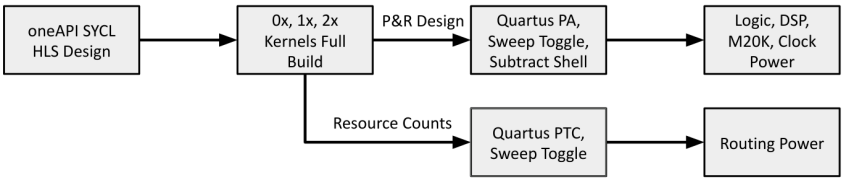


Figure 4.1: Methodology for Quartus-based FPGA power estimation flow.

produce consistent results due to methodological error.

Compute data type. This comparison is done for complex int16 data type. Complex float32, which was used in the previous study, is avoided for two reasons. Firstly, power of a floating point unit on the ASIC side depends heavily on the particular implementation of a well-made floating point arithmetic IP, which may or may not be built into the open source toolchain used. Secondly, adding floating point would increase the overall compute power used by both the FPGA and ASIC. Therefore, it would change the absolute amount and relative proportion of power allocated to compute; however, it would not change anything else such as the trends in the effects of data movement. For these reasons, the kernels are evaluated for complex integer data types. However, it does imply that these numbers should not be directly compared with the previous chapter.

4.2.1 FPGA Power Model

In the previous chapter, the power values used were "wall-plug" power measured directly from the device board power. However, this is not very useful for the comparison in this chapter, as the gross power value does not easily break down into individual power values of computation, memory, data movement, pipelining, and other fine-grained categories. Moreover, the wall-plug power is noisy and captured using a relatively inaccurate board-level power methodology. Therefore, the key

tools for power analysis for this chapter is the post-place-and-route power estimation available from the FPGA vendor EDA tools. In this case, since Altera Agilex FPGAs were used for the initial study and benchmarks, the Quartus Power Analyzer and Quartus Power and Thermal Calculator [75] are used to construct a detailed breakdown of power values. These tools are briefly explained below:

1. **Quartus Power and Thermal Calculator (PTC).** This is the model-based power and thermal estimation tool, where macro values of quantities of each FPGA architectural components like DSP blocks, M20K blocks, registers etc. can be input with certain macro-level characteristics such as toggle rate and routing depth, from which a high-level estimate of power is provided. It does break down the estimate into block power and routing power.

2. **Quartus Power Analyzer (PA).** This is the more accurate post-place-and-route estimation tool where power is calculated directly from the placed and routed design. This flow is able to isolate the kernel system netlists only, and is able to offer a breakdown into computation, logic, memory, and clocking. It does not emit registers and routing values separately.

As routing is one of the primary quantities to evaluate, a combination of the two tools is used in the model: first Quartus Power Analyzer is run on the post-place-and-route design constructed from the HLS flow described in the previous chapter. This is swept across toggle factors as described. Then, for each kernel and each toggle factor, the kernels' resource and activity are analyzed in Quartus Power and Thermal Calculator, which provides a separation between block power and routing power (at default routing settings) as shown in Figure 4.1. The relative fraction between routing and block power is recorded and used to split out the routing fraction from the Power Analyzer numbers. As explained above, not every

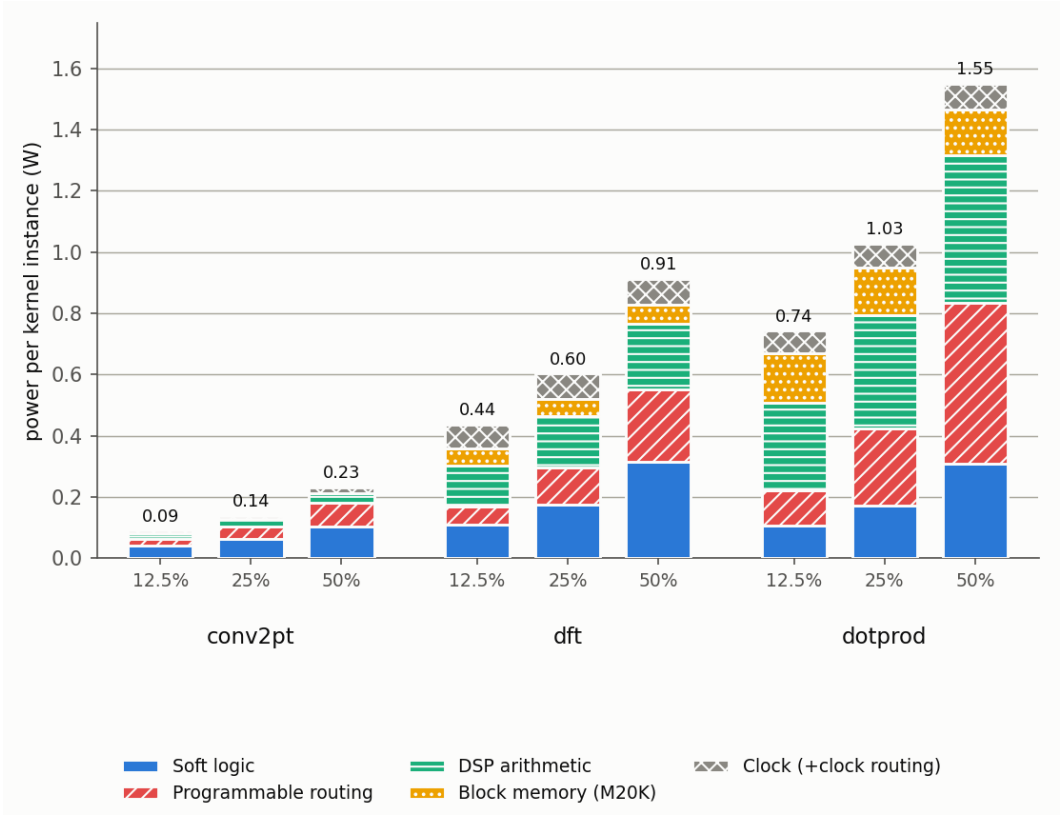


Figure 4.2: Example of FPGA power breakdown using the described methodology.

structure reacts to toggle factors the same way, which is why the ratios vary across kernels and power numbers. Additionally, any data movement or test input harness was removed from this factoring, as that is not present in the constructed ASIC baseline. This was especially relevant for the filter convolution, where a majority of the block RAM usage of the kernel turned out to be a data movement harness for loading weights. Figure 4.2 shows the results of the methodology.

From the above figure, it appears that Power Analyzer is using a lumped model for the compute blocks, and as such the compute power does not grow linearly with toggle rate. This will be different from the ASIC flow, where power estimation scales every net by toggle rate. Power Analyzer is likely the more correct model, as

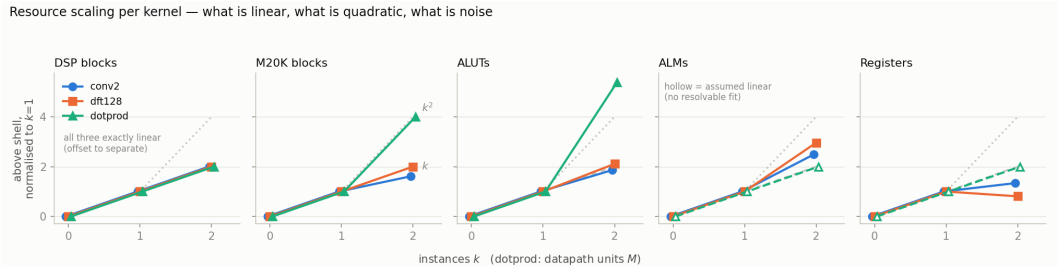


Figure 4.3: Resource counts for kernel instances k=0, 1, 2 HLS builds.

the ASIC model is not an annotated netlist.

A final method of ensuring internal self-consistency is that resource counts and power analysis was run on three different HLS builds containing 0, 1, and 2 instances of the kernel respectively as in Figure 4.3. This allows for the clear isolation of the kernel metrics from the shell and other machinery around the kernel in both resources and power. Interestingly, some quantities exhibit a clear linear relation (such as power and DSP block counts, as expected), making them trustworthy. Other quantities such as M20K block counts are sometimes quadratic instead as an increasing number of parallel reads is sometimes required to feed a linearly increasing number of compute units, representing a limit in data broadcast capability. Other quantities, such as absolute ALM and register counts, varied wildly from build to build and did not exhibit a clear trend. This effect has not been isolated, and perhaps is related to the new configuration allowing some bits to be packed into a hard block, or some variance in timing or usage of HyperFlex registers (all were synthesized to roughly equal clock periods). Therefore, an approximate average was taken for these quantities.

4.2.2 ASIC Baseline Model

There are several difficulties to solve in determining a good ASIC power model for the given kernels. Most importantly, ASICs can be quite domain specialized, and there is no directly comparable ASIC design that can be simply dropped in. Therefore, what is constructed in this chapter is *not an ASIC design or a methodology for constructing an ASIC design.* It is instead an *ASIC baseline* for the given kernel set, focused around answering the question of where the FPGA has kept up or is not far from the given ASIC, and where it falls short. Therefore, it needs to be comparable in the sense explained at the beginning of the chapter but is not required to have all the properties of an actual tapeout- or product-ready design.

To keep the comparison equal, the ASIC model is restricted to the kernel only, and is deliberately kept relatively high level instead of diving into the details of cells and corners.

Kernel Decomposition. Each kernel is first decomposed into operations done per sample. For example, the 2pt convolution kernel does 2 complex-int16 MAC per sample, the DFT-128 does 3.5 butterflies per sample and the 20 beam inner product does 20 complex MAC per sample. For each operation, memory traffic is also calculated, such as delaying values, reading coefficients, or weight bank reads for inner products (as the weights are relatively stationary as the samples are streaming past). These are then scaled by the sample rate, clock rate, and instance counts of the measured design, for example the SOAP design runs at 1.25 GS/s for 128 subbands, scaling up all of the required operations.

Measurement of Key Blocks. Simple implementations of the key blocks, such as the 18-bit complex MAC (akin to the Agilex variable-precision DSP in int16

Operator / memory structure	Array	Per sample	At 1.25 GS/s
conv, 2-tap polyphase	148 instances, 32 b complex-int16 words		
complex MAC	—	2 op	370 Gop/s
delay line	1 KB	64 b	11.84 Tb/s
coefficient read	2 KB	64 b	11.84 Tb/s
memory subtotal		128 b	23.68 Tb/s
DFT-128, radix-2 streaming	148 instances, 32 b complex-int16 words		
butterfly	—	3.5 op	647.5 Gop/s
delay-feedback lines	14.3 KB	448 b	82.88 Tb/s
twiddle read	0.5 KB	112 b	20.72 Tb/s
memory subtotal		560 b	103.60 Tb/s
inner product, 20 beams	64 instances, 32 b complex-int16 words		
complex MAC	—	20 op	1600 Gop/s
weight bank	10 KB	640 b	51.20 Tb/s
memory subtotal		640 b	51.20 Tb/s
Internal data traffic			**178.5 Tb/s**

Table 4.2: ASIC kernel bandwidths used for scaling the energy values.

mode) are sent through a full RTL → GDS flow using the OpenROAD toolchain. Then, any moved bits are scaled according to Table 4.2.

This chain uses Yosys [94] to synthesize using the ASAP7 PDK, then OpenROAD tools to plan, place, perform clock tree synthesis, and carry out a detailed routing step [92]. Some noteworthy implementation details are given below. Firstly, the complex MAC uses `-booth` flag in synthesis to decrease power by 30% compared to Yosys' default mapping. ORFS supplies a good map for adders, but not for multipliers, so the default arguments produced bad multiplier structure. (This is still not comparable to a handmade design). The DFT butterfly is a similar 3-stage pipeline. However, the synthesis flow does support a retiming argument via `abc`, which does not work with this RTL. Every state element has an async reset, causing the retimer to create erroneous registers. Power is then reported from this using OpenSTA using a vectorless implementation as explained above in a manner akin

block	function (precision)	energy (pJ/cycle) 12.5 %	25 %	50 %
(a) operator blocks				
`bf_ipe`	complex MAC (int18)	23.52	41.12	70.40
`bf_ipe` Booth	complex MAC (int18)	15.98	26.08	45.28
`bf_bfly16`	butterfly (int16)	29.76	54.24	70.08
`bf_cmul16`	complex multiply (int16)	13.63	26.08	49.60
derived: add/sub/pass	`bfly16`−`cmul16`	16.13	28.16	20.48
(b) pipeline & distribution primitives				
`reg_w18_d1`	reg, 18 b	0.032	0.041	0.058
`reg_w36_d1`	reg, 36 b	0.063	0.081	0.116
`reg_w36_d4`	reg, 36 b×4	0.347	0.398	0.498
`reg_w96_d1`	reg, 96 b	0.192	0.240	0.333
`fout2_w36`	fout 1→2, 36 b	0.142	0.174	0.238
`fout4_w36`	fout 1→4, 36 b	0.160	0.200	0.280
`collect4_w36`	collector/steer, 4→1, 36 b	0.579	0.818	1.000
`collect20_w96`	drain, 20-entry, 96 b	7.712	10.368	11.360

Table 4.3: ASIC operator library with all blocks closing timing.

to the FPGA vectorless power analysis. The placed and routed blocks are pipelined and clocked to reach the two target clocks of 625 MHz and 1.25 GHz. This pipelining and other details are internal. The results of this methodology are listed in the block results of Table 4.3.

An interesting takeaway from the above is that the pipelining and data steering components are very clock power-heavy, while the computational components are not. Pipeline registers are 75-78% clock power with similar numbers for the other data movement components, against 3.2% for the MAC. Eliminating these would be a focus of the next chapter. The operators above are composed into kernels at a high level in a pen-and-paper manner, as only enough resolution is required to compare to the FPGA power numbers, and not for a real ASIC design. This is also due to ASIC-optimized RTL and design flow being very different from FPGA-optimized RTL and design flow. The block-level structure of dotprod is shown below.

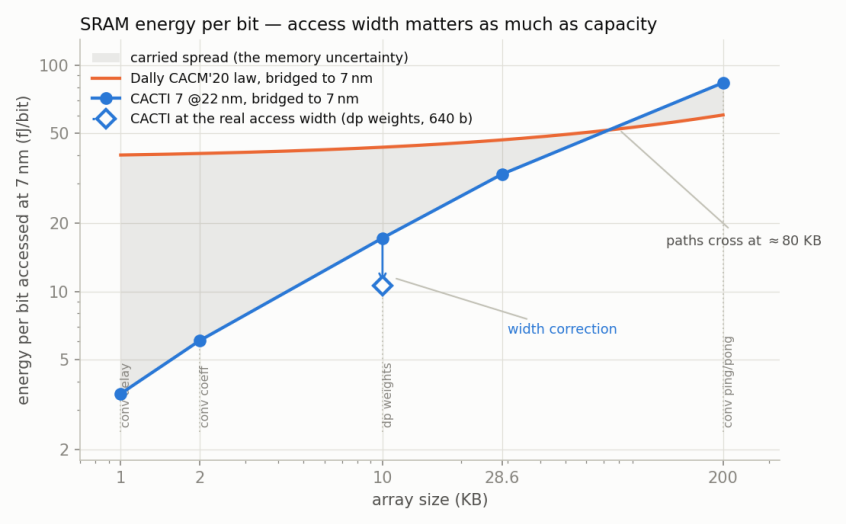


Figure 4.4: SRAM energy model showing the two published paths ([96] and [8])

The Cost of Memory Accesses. Based on literature surveys, it is possible to estimate the pJ per bit for SRAM access as a function of its size [8]. Additionally, access width is also considered using the CACTI models [96] scaled [97] as in Figure 4.4, which has a small but not inconsiderable effect on SRAM power. This is important because, as data transport kernels are read-heavy, a wide access size is one of the optimizations explored in the next chapter. The effects are shown in the graph below. A structural advantage for the ASIC here is that the SRAM can be assumed to be distributed and banked exactly as required for the application, instead of needing multiple pre-built banks as on an FPGA. This allows the ASIC to be considerably more efficient in SRAM usage as it does not have to pay for this fragmentation.

Assembly. The calculated values are assembled into a final design following the same dataflow layout as the FPGA design, and scaled by the required activations per second (controlled by sample rate) and the number of instances (controlled by

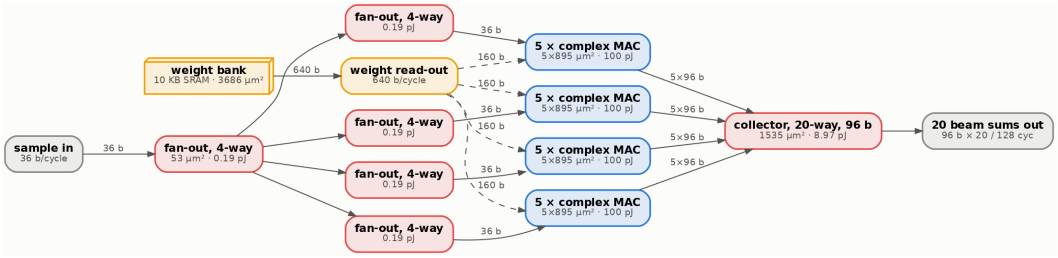


Figure 4.5: Block diagram structure of dotprod (inner product) kernel. Pipeline buffers and clock not shown.

required throughput). An example of the inner product block diagram is shown in Figure 4.5. Many low-level details have been under-represented here, such as an explicit design-level clock tree. The clock net power in the ASIC is captured at the endpoint level as part of the CTS (clock tree synthesis) step when processing a given operator. Therefore, the clock power only counts the power inside each operator and does not account for the global clock structure across the ASIC, as there is no "real" ASIC design to measure. Wire lengths are estimated as just the sides of the given blocks, with SRAM geometry estimated through CACTI at 22nm (this is left as is for a conservative estimate).

4.3 Results

Ablative Analysis, Reference Values, and Contributions of Each Modeling Stage. The impact of the above modeling methodology significantly increases the reported ASIC wattage from a naive 1.20W as quoted by just following the computational ops energy without any sort of other modeling, with a "no regs and no wires" caveat (again, showing the intrinsic overhead of data movement). From the literature, a range of uncertainties is derived that accounts for the uncertainty of the open source toolchain and the large published spread in SRAM energies.

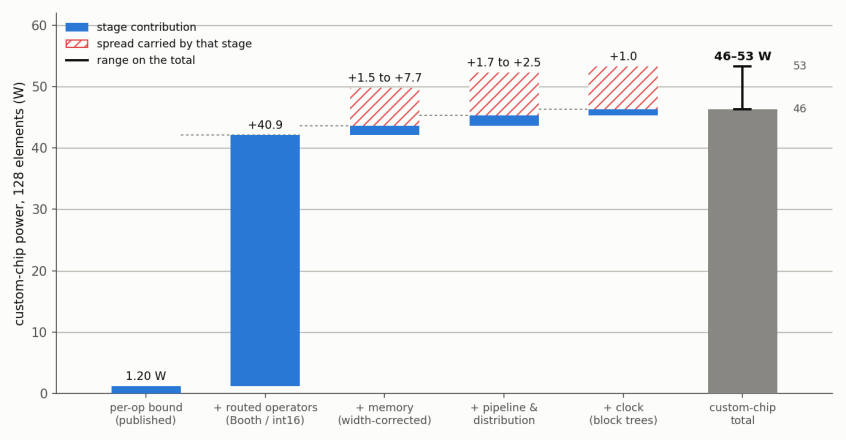


Figure 4.6: Contribution of each ASIC modeling stage to the final derived power.

However, as the values of interest are in the final breakdown of power across all the components, individual variances are not that damaging to the final argument. The overall average power output for inner product is 46–53W, and it decomposes as memory (1.5–7.7W), the pipeline and distribution account for 1.7–2.5W, and the clock accounts for 0.8W as shown in Figure 4.6. No correction has been made for the open source vs. commercial flow as that is highly design dependent, as such, the results have been reported as is.

	FPGA (W)			**ASIC (W)**		
Component	12.5 %	25 %	50 %	12.5 %	25 %	50 %
Soft logic	29.0	46.5	81.9	—	—	—
Routing	19.5	39.8	80.0	—	—	—
Arithmetic	41.4	52.1	67.0	39.5	67.4	120.5
Block memory	18.4	18.4	18.9		1.5–7.7	
Data movement	*inside logic and routing*				1.7–2.5	
Clock	17.2	18.8	20.7		0.8	
Total	**125.4**	**175.6**	**268.5**	**43.5–50.5**	**71.5–78.5**	**124.5–131.5**
Ratio				**2.5–2.9×**	**2.2–2.5×**	**2.0–2.2×**

Table 4.4: Per-component power at 128 elements, all three kernels (DFT-128 at the datapath precision, complex int16).

Component Power Comparison. In the final accounting, from Table 4.4 the FPGA vs. ASIC ratio ranges from 2.5–2.9× / 2.2–2.5× / 2.0–2.2×; exemplar 268 W FPGA vs 125–132 W ASIC at 50%. It is expected that the gap shrinks with increasing toggle rate as the FPGA's structures (especially arithmetic) scale much more flatly with toggle. As mentioned above, this is a methodological difference in using different kinds of power models, and requires simulation-annotated activity to bracket properly. Meanwhile, data movement scales linearly in both. While large in absolute terms, this is smaller than the gaps often cited in literature when FPGAs are evaluated using soft logic, showing the efficacy of hard blocks. It is within range of other reported literature when the comparison is done with hard blocks, such as ≈ 7.1x [27]. However, the key novelty here, which is the decomposed comparison, works out at such: the granular decomposition shows that arithmetic is probably not a good attack surface for FPGA improvement (as noted above, FPGA blocks are more versatile, as well as the methodology being inaccurate, the only conclusion to be drawn here is that they are within range of each other). The highest wins lie in logic, routing, SRAM memory configuration, and the clock net, which can be addressed in priority order in the next chapter.

The results of the analysis above are presented as in Figure 4.7. As predicted, the toggle rate has a considerably large influence on the overall power of the chip, although as stated above, the item of interest here is the relative breakdown between the ASIC and the FPGA.

4.4 Analysis

For the FPGA, it can be seen that the programmable logic and interconnect are the single largest burners of power in the system, far in excess of the other quantities,

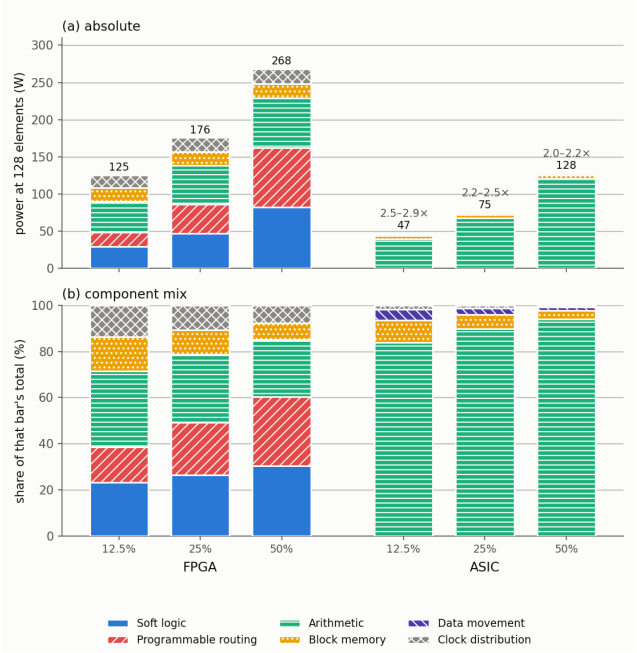


Figure 4.7: FPGA vs ASIC power breakdown comparison.

but most things scale roughly linearly. The clock distribution is the only constant, which is expected as it toggles every cycle regardless. The same is true on the ASIC side as expected. This is a point that is perhaps not as commonly seen in GPU and other accelerator architectures, but does rear its head when examining FPGA-style architectures in this context.

The following is the trend, per activity level, of the comparison between the FPGA and the ASIC. From this we can make some observations:

1. Falling Ratio. From 12.5% to 50% activity, the overall ratio falls from 2.9× to 2.2× (high range). This is because the ASIC blocks in ASAP7 flow are more sensitive to toggle rates than the power model used by Quartus Power Analyzer. The gap is widest at low activity, which is encouraging, as most real-world signal data have some correlation at least.

2. The relative fraction of components changes substantially. At 50% activity, the difference is mainly due to soft logic and routing, which shrink rapidly in proportion, while the fixed terms do not. The more expensive clock network and the block-memory penalty account for nearly half the difference at quieter activity factors. This suggests that while a headline piece might be to look at routing, the clock and memory organization are just as important given that most data is somewhat correlated.

3. The penalty of using a pre-sized memory structure instead of a custom structure is nearly constant, which is understandable given that the memory banks are being read every cycle, which costs an access regardless of data characteristics.

4. The arithmetic blocks and the DSP blocks of the FPGA may have very different power, but they should be judged to be "within range" and therefore not a significant contributor to the difference. While it may be tempting to conclude that this is because the FPGA DSP blocks are essentially ASIC computational units, they are not equally capable - FPGA DSP blocks are hand-tuned variable precision units capable of a lot of different kinds of operations, while the ASIC block is a standard cell synthesis on an open source toolchain which implements exactly one operation.

4.5 Conclusion

In summary, the measured implementation supports the two statements that this chapter sought to demonstrate. The FPGA-ASIC gap at the kernel level is only about 2-3x, not the 14x of classical understanding [27], due to modern features

such as the presence of hard blocks. Secondly, the major cost does not come from computation; rather, it comes from moving data through the programmable interconnect and logic, from clocking the deep pipelines that programmable routing makes necessary, and from the ill-fitting replicated memory blocks that the fabric's fixed architecture forces. At 50% toggle, soft logic+routing 161.9 W of 268.5 W (60%); clock 20.7 W vs 0.8 W; memory 18.9 W vs 1.5–7.7 W. Therefore an architecture specialized for data transport must attack the interconnect and logic terms at high activity, and the clock and memory-access terms at lower activity, but all four are critical. In the next chapter, these tendencies will be used to inform an optimized device. An additional finding was that a significant portion of the FPGA kernel RTL was actually just data mover units (such as LSUs and rings) from DDR memory. A true data movement fabric does not require such structures, as memory flow is not a considered usecase. This has significant implications for the fabric of an optimized FPGA, where the need to support such data movement structures is much smaller.

Chapter 5

Flow FPGA: Data Transport Optimized FPGA

5.1 Introduction

This chapter proposes Flow FPGA, a novel class of FPGAs that is built for data transport computation. This is a streaming-only device where data flows from one side of the fabric to the other, with computations happening to it along the way. Flow optimizes towards the domain using three main pillars, namely, an extensive set of hard blocks tailored towards the domain to achieve high efficiency, a mix of memory blocks for the appropriate data bandwidth, and minimizing the distance that data has to travel from IO source to IO sink. The Flow FPGA is not meant for traditional sea-of-gates ASIC-emulation, infrastructure offloading, or for general acceleration workloads as is typically done on FPGAs and FPGA SmartNICs today. Rather, it is meant to support multiple instances of the short, high-IO low-compute feedforward pipeline archetype that is identified with data transport compute in

Chapter 1.

Along with the computation, the usage modality of Flow is also specialized. It is not a compute-offload accelerator like many GPU, ASIC, and FPGA accelerators today, where blocks of work items are transferred by a CPU from host memory to accelerator device memory, and a go signal is given upon which the accelerator processes the work items. That requires an architecture that supports deep control integration with the host, memory-like traffic and memory-like interfaces obeying deep request-response pipelines and an address-based streaming paradigm (such as AXI-MM). Additionally, supporting this usage requires memory interfaces and load-store units, which are additional burdens on the fabric. Instead, Flow FPGAs are purely streaming devices that are integrated along with other traditional devices on data streams, with the intention of offering inline computation that would not be well-suited on a traditional device with a deep data-reuse oriented cache hierarchy. This allows for efficiency and performance through specialization for this particular domain.

In this chapter, the Flow FPGA model is built up starting from a simple resource model that takes as input the basic parameters of data transport computing (namely, input throughput, device cycle time, latency bound, constant storage, access widths, computational composition of required operations). This is then used to propose a long, thin FPGA with wide parallel IO along the long edges, creating a shallow path for a "computational pit stop" and making integration simpler. The chapter concludes with design sweeps for a suite of transport kernels showing FPGA structural improvements over a baseline architecture using the suite of data transport compute kernels from the identified application domains. For the SOAP kernels from Chapters 3 and 4, these structural improvements are then translated

into power improvements and compared, substantiating the second claim of the thesis that the gap to ASICs can be recovered by choosing the right design tradeoffs. The question of integration of Flow into computing systems will be explored in the next chapter, driven by the wide parallel streaming interface.

5.2 Motivation

Restated briefly, the essence of the second claim of this thesis is that FPGAs are already good for data transport computation, and with specialization and domain-specific optimizations can significantly close the gap to ASICs, making them redundant. The previous chapters have demonstrated the suitability of existing FPGAs with existing methodology (along with a critique of the current crop of FPGAs for such use-cases, using Altera Agilex FPGAs [73]), as well as quantifying the gap to an ASIC baseline. This chapter, proposing Flow, therefore completes this picture by actually constructing and evaluating possible versions of the aforementioned specialized and domain-specific FPGA. All the salient properties of the Flow FPGA (particularly in how it differs from a generic device) stem directly from the data transport computation definition, analogously to how multi-level caches are developed and integrated into CPUs and GPUs, falling naturally out from the observation that the programs that are to be run exhibit significant spatial and temporal locality [1].

Importantly, Flow is not a novel FPGA in the sense of inventing new kinds of connectivity or fabric structure. It is instead an extreme parameterization of today's relatively regular and modular FPGA fabrics. Here, architectural levers such as aspect ratio, soft logic provisioning, and embedded

hard blocks make the correct tradeoffs for data transport computing as opposed to general purpose computing. A dedicated vendor could build a Flow fabric today by choosing the same tradeoffs. This follows from a long tradition of vendors deploying FPGAs that are optimized towards a certain domain, such as FPGAs for the ASIC emulation market that are almost entirely soft logic [98].

There are other relevant considerations include the reasoning behind using an FPGA in the first place, which is elaborated on in this chapter. The key reason, other than already good efficiency (and even better with Flow FPGA), is the bit-level flexibility allowing design customization that is difficult on coarser fabrics, as it is here a relatively *low-overhead* method of adding programmability, control, and customization to a sea of hard block accelerators, instead of requiring an instruction-driven interface. Secondly, in any application, the FPGA can implement any residual design not captured by its hard block in soft logic or through creative combinations thereof with hard blocks. This is somewhat quantified in this chapter by the variety of applications that the Flow fabric and model can support from a relatively simple list of primitives. The tradeoff for this is naturally a more difficult development process based on RTL design or HLS, which is an orthogonal problem assumed to be acceptable for the purpose of this dissertation. Additionally, embedding the sequencing and control of these hard blocks in FPGA fabric makes them readily programmable with FPGA toolchains, such as VTR [52] which is what will be used in this chapter. This sidesteps the common ASIC requirement for needing custom software toolchains for programmability.

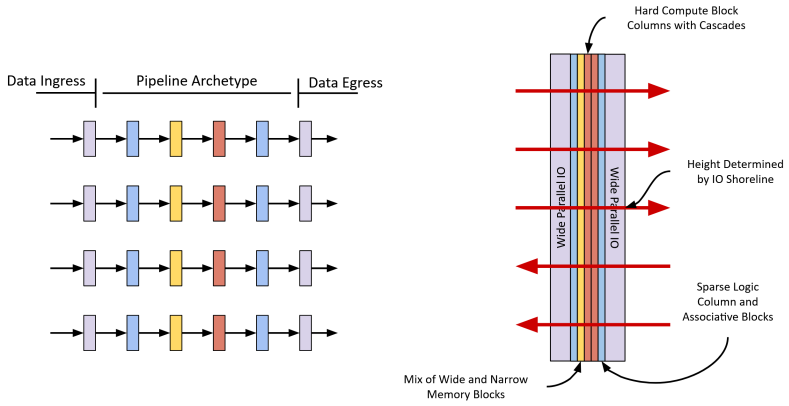


Figure 5.1: Flow FPGA fabric constructed out of conceptually stacked pipelines.

5.3 Flow FPGA Characteristics

The key change that has been made in Flow FPGA is the trend towards three extremes as seen in Figure 5.1: extremely thin aspect ratio, to promote short IO-to-IO paths; extremely high provisioning of tailored hard blocks, compared to a more general purpose fabric such as Altera Agilex [73]; and memory and connectivity that is tailored to the application. The intent is to move Flow FPGA away from a general purpose fabric while retaining a configurable amount of flexibility. Additionally, the interfaces are streaming only, with no memory request-response IO, which decreases the pressure on the fabric for supporting structures such as load-store units, ring buffers, and other data distribution paraphernalia that are not required for streaming data flow.

5.3.1 Stacked Pipelines and Shorter IO

Simple reasoning readily derives the basic design. The core archetype of the data transport kernel is the short bandwidth-reducing (or compute-reducing) pipeline, where, due to bounded reuse and bounded latency, it is unlikely for the pipeline

to suddenly amplify traffic or compute demand in the middle. For efficiency, it is assumed that this pipeline is made largely of hard blocks. As seen in the DARPA-SOAP application, it is common for there to be multiple parallel flows executing at once (in that example, multiple parallel subbands and polyphase filters at once). To keep the IO-centric philosophy, it is therefore natural to stack these pipelines up along the IO direction, creating a long and thin fabric tile as in Figure 5.1. This is the basic idea behind the Flow FPGA fabric tile.

A consequence of the above is the reduced distance from IO ingress and egress ports, and the fabric-friendly wide parallel IO. This simplifies and makes the system more efficient in two ways. Firstly, as there is simply less distance to move, it reduces the wiring required to get there. Secondly, it eliminates the need for both hard and soft data movement structures, such as long FIFOs, LSUs, or NoCs, because the input or output IO ports are located nearby. Together, this creates a defining IO-centric feature of the Flow FPGA fabric.

The extremely thin aspect ratio (just a few columns, compared to tens to hundreds in commercial fabrics) does impose a few limitations. For designs where the short pipeline archetype fits directly across the fabric, this works well. However, if the pipeline does not fit across, it must percolate vertically instead. Due to the extremely low soft logic provisioning, this reduces the available routing to the IO along the vertical length of the pipeline, effectively cutting them off and decreasing the overall bandwidth that can be handled by the device. To counteract this, additional columns can be added to the Flow device to support longer pipelines to free up the IO column again. This does not have a very high cost in a pipeline, but would increase the cost for kernels that do not need the deeper pipeline.

5.3.2 Hard Block-Centric Fabric

The Flow FPGA fabric optimizes for the problem domain using a larger proportion of hard blocks than is typical in an FPGA. Two key changes are presented here: firstly, a slightly larger than usual compute block with multiple outputs for directly performing complex operations and DFT butterflies. This increases wiring cost of the tile, but subsumes a lot of soft logic and multiple hard block usage. It is characterized using Chapter 4 OpenROAD results. Secondly, a few specialized blocks have an outsize impact on reducing soft logic and routing, as they encompass key fundamental operations such as associative lookup. These are embeddable in a parametrizable way inside soft logic columns.

Hard Block Provisioning. Using a simple resource model, it is possible to, with relative accuracy, predict the hard block demand for data transport kernels, as their bounded reuse and bounded latency make this relatively simple (soft logic demand is not nearly as simple and is a case for FPGA flexibility). Therefore, a Flow FPGA fabric can be provisioned via a design space sweep that is informed by the model and generally produces good results. An extreme specialization often results in block densities (hard block vs soft logic) greater than 2x of Agilex 7, in that Flow FPGA instantiations often have more than twice the hard block provisioning relative to soft blocks that the commercial Agilex fabric has. However, design sweeps can also be run for a mixture of workloads.

Associative Lookup Block. CAM and associative structures are common in data analytics workloads, and the most important structure in the aggregation kernel is the associative lookup. Therefore, Flow FPGA adds a 64-entry CAM block to the fabric with approximately a pin-equal interface to a RAM block, requiring minimal

perturbation of the fabric. The CAM block only maps indices and relies on a nearby RAM block for actual storage. It has a search port and an insert port, to mimic the RAM block's read port and write port. Larger associative lookups can be built by binding CAM blocks together. An important consideration here is the library of macros built for the new hard blocks, which at the moment only express individual blocks. A CAM macro could, in theory, offer arbitrary key size and depth by chaining CAM blocks, but at the moment these must be wired up by hand. CAM blocks are placed within logic+CAM columns which are near a memory column.

Compute Blocks With Additional Output. As DFT is a common operation for streaming signal processing workloads, a set of compute blocks with extra outputs to express butterfly operations was also included (with a+b and a-b outputs in butterfly mode). Without this, in DFT-style kernels, the wiring engages in a lot of "thrashing" (analogous to cache thrashing) where it chaotically moves between soft logic and compute block columns. It may be argued that this is simply hardening one of the required kernels. However, this is the major primitive of a DFT kernel and so is useful for any streaming DFT implementation of any size. Additionally, it is still a compute block and as such can participate just like before in more compute-dense kernels like inner product. As the overall fabric is being made more sparse, qualitatively it should not be difficult to absorb the extra output pins of a given column. Quantitatively, this block is sized by taking a ratio with the equivalent blocks from the ASIC study in the previous chapter, resulting in a conservative 2.5x size increase. As such, its fabric cost has likely been overestimated.

Reduced Soft Logic. A key feature of the general purpose FPGA is the generous amount of soft logic, which serves both to implement designs that cannot be

implemented any other way (i.e. not in hard blocks) and also serves an important performance role. Soft logic and its associated routing are often called upon to implement high-performance structures that cannot be done flexibly any other way, like, for example, a few constants stored in registers very close to a compute block's inputs. Additionally, they also provide flexible routing any which way as required by the fitter to place and route the design for high performance.

However, this is resource-consuming and energy-consuming, as it was by far the most dominant term in the previous study. Therefore, for the purpose of optimized data transport compute, wherein most of the active operations are meant to be in hard blocks with soft logic performing only basic control, sequencing, and the very occasional register or high-performance structure, the soft logic provisioning has been decreased considerably. The positive example kernels (data transport compute kernels) fit with some margin, while the negative general-purpose and ML compute examples from the Koios benchmark either fit poorly or not at all (despite sufficient hard block resources).

5.3.3 Data Access, Routing, and Connectivity

Mixed Memory Requirements. The data transport compute pattern's memory demand often falls into two very distinct buckets: firstly, driven definitionally from the data transport scenario, these kernels often have long-lived operands that are largely read-only with most compute blocks needing to read them per cycle, which is a large parallel access bandwidth with not a lot of depth required. Secondly, general-purpose buffering, stream buffering, pipeline stages, and table-based kernels such as aggregation require deep memories to express the relatively thin table structures. Therefore, Flow can provision both an Altera M20K-style general-purpose deep

RAM block (from Koios [99]), and a few shallower-but-widely-banked SRAM blocks scattered for mostly-read operands.

Hardened FIFO and Steering Blocks. Two common primitive operations in the set of kernels were FIFOs for elastic buffering and streaming connectivity, and aggregate data steering structures such as comparators and merge. These derive naturally from the common streaming primitives of pipe, fork, and merge, with merge being particularly difficult in soft logic due to the muxing, and the pipe having control logic overhead. The fork distribution is relatively simpler. Therefore, hardened FIFO and steering blocks are scattered throughout the logic columns, further decreasing the logic density. These absorb the soft logic resources required for *common streaming data routing*. In some ways, these blocks are a proxy for registered coarse routing, which cannot be expressed in VPR at the moment [70]. The absorption of FIFO pointers, control signals, and backpressure along with simpler FIFO storage that does not require external address input additionally makes this common structure much more efficient on Flow FPGA fabric, and further reduces the role of soft logic.

Fabric Routing: Out of Scope. Several routing-level optimizations were tried, such as changing F_c, changing the mixture of long, medium, and short wires, and changing the distribution of wires. However, at this architectural level, this had very minimal effects, at best, on overall placement or power. Additionally, several circuit-level changes such as high Vt cells and lower swing drivers on the wires were considered, but could not be implemented in the chosen tooling (needs additional tools). Therefore, routing has not been considered as a key feature of Flow fabric, although it is likely that in an actually engineered situation, routing optimizations

can make a significant contribution towards lowering overall power. This is because the fabric's overall usage pattern is roughly known, reducing the need for general-purpose wiring that tries to serve every use case well. Additionally, the clock net can also be optimized to ripple along with the kernels instead of using (for example) a balanced H-tree. It also has to clock fewer items, as the overall architecture is leaner. These were also not possible to test with the available tooling and were therefore considered out of scope. Therefore, from an engineering standpoint, the Flow fabric could be considerably better than what is demonstrated here.

5.4 Related Accelerators

With the recent boom in AI and the widespread recognition that data movement is a key bottleneck in many applications, there has been a corresponding explosion in commercial accelerator architectures which combine streaming, compute blocks, and memory blocks. On first glance, this puts Flow FPGA in direct contention with these architectures, however, there are major differences in both form and function, on every major column.

Streaming Dataflow Architecture Comparison. The most important difference is that the other architectures are host- or memory-fed ML accelerators, which are not built to take purely upstream-controlled streaming input. This is not what the data transport computing scenario requires, as without support for direct streaming to compute and a low-overhead control scheme, these architectures fall into the same pitfalls as the GPU and TPU from Chapter 1. Flow FPGA is meant to be as simple as possible while keeping FPGA benefits, as the problem that Flow FPGA is meant to solve is relatively bounded and usually does not involve that much

	Flow (this work)	**Groq [46]**	**Tenstorrent [48]**	**Cerebras WSE [49]**	**SambaNova RDU [47]**
Geometry	1-D ribbon	1-D slices	2-D mesh	wafer-scale mesh	2-D checkerboard
Control	none (wiring)	temporal ISA	MIMD RISC-V	dataflow-triggered	dataflow config
Interconnect	hardened cascades	fixed lanes	packet NoC	packet mesh	word switches
Granularity	word + bit	tensor slices	tensor packets	word dataflow	word/pattern
Memory	small, scattered	weights-resident SRAM	∼1 MB/core	tiny, per-tile	interleaved, tiered
Primary input	edge streams	host/memory	host + Ethernet	host	host + memory
Determinism from	the wiring	the compiler	not a goal	arrival order	the mapping

Table 5.1: Flow FPGA compared with four commercial dataflow accelerators.

computation. The other architectures are "big" devices with compiler control, wide and deep memory interfaces, vector engines, mesh networks, and other features required only when the computational sequencing and memory flow becomes relatively complex, and there is a lot of reuse and re-reading of memory elements. Table 5.1 breaks down the difference in more detail, focusing on architectural geometry, control scheme, interconnect, granularity, memory, input, and determininsm (order). As seen from the table, the compared architectures, in their default instantiation, are high-overhead for simple streaming computations of the kind expressed here, and are therefore not suitable. Cerabras [49] and SambaNova RDU [47] are closer to Flow FPGA than the others, in that they have dataflow-triggered or spatial dataflow control, with more flexible memory, and it is possible that a very minimal example of their architecture would be appropriate for data transport compute, in a vacuum. However, they then would not have the Flow FPGA's bit-level flexibility

and would become essentially CGRAs.

Network-on-Chip. The question of whether to use a NoC (Network-on-Chip), as is key in data movement FPGA devices today, was already addressed in Chapter 1. However, it should be restated here in the context of Flow FPGA as it is one of the main features of modern datacenter FPGAs such as Altera Agilex M-series [55], AMD Versal [53], and Achronix Speedster 7t [56] devices. As already covered in Chapter 1, a NoC is a structured network with very high bandwidth made for high throughput long-haul data transport from one part of the chip to another without unduly disturbing the fabric underneath. It is an architectural specialization towards high-bandwidth packetized data transport. The purpose of the NoC is to enhance connectivity to far-off regions, supplanting the need for very long wires. A NoC is analyzed and designed as a network, with routers, ingress and egress ports, request-response facilities, virtual channels and buffers etc and supports typically a packet based interface and implementation. The NoCs in the above mentioned FPGAs are often configured to carry memory or Ethernet traffic into the fabric, acting more as a hardened load-store spine than a generalized transport in all but the Achronix Speedster 7t.

In Flow FPGA, this modality is both irrelevant and orthogonal to how the device is meant to integrate into a system. There is no explicit need for long-range transport, as either the data flow enters the fabric through the wide IO interface, immediately gets computed on as it goes across and exits immediately, or it percolates down the devices and exits eventually. There is no long-range distance problem to be solved. However, for larger instances, or when combining multiple Flow FPGA tiles together, a NoC may be used, without changing the objectives of Flow FPGA. Therefore, it is an orthogonal decision and likely a good combination under the right

circumstances.

5.5 Sizing a Flow FPGA Device, Resource Model

As the data transport problem is a constrained one, in that setting a few variables tends to define what the structure of the problem will be, it is instructive to try to create a very simple model that is only dependent on the generic problem variables. This simple model can then predict certain device parameters, such as the number of DSP blocks, allowing the design sweeps to start from an already-decent design. Examining the problem, there are four resource-relevant variables that can be extracted. These variables are a property of the *system* (the precise scenario that the transport kernel is opreating in). They are,

1. The arrival rate or input throughput λ
2. The reuse per operator class R_c
3. The allowable latency L
4. the working-set size S

Using Little's Law [90], these can be straightforwardly converted into resource budgets. The table Table 5.2 provides a simple mapping that does well in all the examined kernels for everything but the soft logic, which is natural.

The soft-logic covers uses such as controls, handshake, formatting, miscellaneous routing, and other things that are difficult to account for in a simple resource model. This is then the "FPGA-ness" of the solution, as the other resources could also have been done on ASIC, but this uncertaintly is exactly what the programmable fabric exists to solve. Essentially, the right procedure is to harden what can be determined easily here, and leave the rest to the soft logic.

Rule	Meaning
(1) $H_{\text{lane}} \geq \lceil \lambda w/p \rceil, \quad H = KH_{\text{lane}} + 2$	Edge pads (p per border tile) bound lane height; K lanes stack
(2) $N_c = \lceil \lambda R_c \rceil$	Operators per hard class c at utilization 1
(3) $B = \lambda \max(L - L_{\text{d}}, 0)\, w, \quad N_{\text{fifo}} = \max\big(\lceil B/C_f \rceil, \lceil w/w_f \rceil\big)$	In-flight items $\Rightarrow$ buffer bits; capacity- or width-bound
(4) $N_{\text{mem}} = \max\big(\lceil S/C_{\text{blk}} \rceil,\ \lceil bw_S/w_{\text{blk}} \rceil\big)$	State S at read bandwidth bw_S: capacity- or port-bound
(5) $N_{\text{spine}} = \lceil N_{\text{M20K}}/(8\lfloor H_{\text{lane}}/20 \rfloor) \rceil$	Memory columns per lane; one spine supplies 8 M20K per 20 rows

Table 5.2: Flow FPGA sizing guidelines using Little's law $N = \lambda L$ for a stream of λ items/cycle at w bits/item.

5.5.1 Methodology

Stock VTR for Architecture Capture. This study is carried out with stock VTR/VPR, an academic FPGA toolchain [52] that allows one to define their own fabrics, making it extremely useful. Stock VTR 9.0.0 was used, for results reproducibility. Additionally, as this is an architectural study and not a devices and circuits or CAD one, any modification to VTR would have been indefensible.

The capture starts from the stock Koios 22nm VTR capture [99] and uses that as the base VTR XML. This capture has zero R and C on wires and switches, therefore VPR's power estimation does not work. Therefore the power model has to be analytical. A fabric generator script accepts a column plan and device height and assembles the VTR fabric from the Koios base by replacing `<layout>` blocks, setting routing configuration, swapping tiles and adding block definitions and cascade directs as necessary.

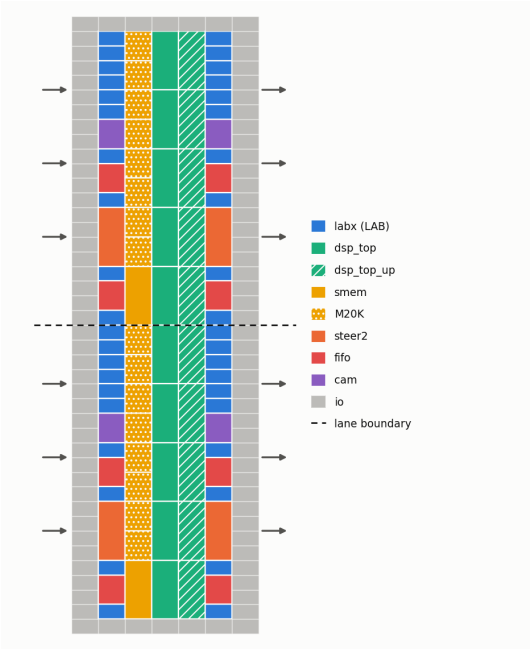


Figure 5.2: An example Flow FPGA fabric showing the new block types.

Hard Blocks. VTR blocks have three pieces, a `<model>` (the netlist primitive), a `<tile>` (the physical dimensions, pins, and so on), and a `<pb_type>` (the logical block, modes, timing, etc). Additionally, there are memory-class primitives (which slice and pack data) and regular hard blocks (DSP, CAM etc.). The blocks used are given in Table 5.3, with a small description.

Timing is copied from the base architecture's memory primitives. The area is always in units of an existing tile. An example visualization of Flow FPGA fabric is in Figure 5.3.

Cascades and VPR Workarounds. To provide the compute blocks with dedicated data paths, hard block-to-block `<direct>` entries are used. However, with the strip geometry of Flow, a unidirectional cascade often ended with data crossing large swathes of the fabric especially when multiple columns of compute blocks were

Block	Origin	H	Area	Function	Used by
`labx`	Koios	1	27 905	plain LAB (soft logic)	all; KAN runs on a soft-only strip
`dsp_top`	Koios+new	4	636 147	mult–add, cmul, or butterfly mode; 64 b cascade, 27 b scan	FFT (butterfly); beamformer, FIR (cmul)
`smem`	new	4	137 668	64×320 single-port stream store; 32 b column write	beamformer
`memory`	Koios	2	137 668	M20K	FFT, aggregation
`cam`	new	2	27 500	64×32 content-addressable	aggregation
`steer2`	new	4	154 795	2×2 steer / 2:1 merge; 36 b ALU; $\leq$64-deep delay; ready pins	MSPM compactor
`fifo`	new	2	41 672	64×64 b hardened FIFO (pointers, flags, backpressure)	all stream boundaries

Table 5.3: New or edited block types in Flow, drawn from the kernel set.

used. Therefore, up- and down-cascades were added to allow both side connectivity. Initially, an effort was made to do this as a modification to the default compute block, however VPR rejected a block with two `<direct>`s, and creating two logically separate blocks did not help either as the placer would simply pick one always. Therefore, if one would like to use the cascades, two separate compute block macros exist, one up and one down cascading, which are placed interleaved. Partitioning between the two has to be done manually in the RTL, but this is a toolchain limitation and not a technique limitation. Additionally, commercial fabrics have support for cascading as well [73], making this more of an equalizer than a true improvement for Flow FPGA.

Hard Block Library The Flow FPGA hard block macro library currently exposes each block directly without any macro-level configuration or the ability to automatically compose designs out of multiple blocks as shown in Table 5.4. This is an orthogonal feature that would go into the design of, for example, a macro library. It would be quite useful for, for example, CAM blocks, where multiple blocks could

Block	Inputs	Outputs
`dsp_top`	`dsp_I1`, `dsp_I2` 2×64; `chainin` 64; `scanin` 27; `reset`	`result` 74; `chainout` 64; `scanout` 27
`dsp_top_up`	as `dsp_top`, cascade direction flipped; no butterfly mode	
`steer2`	`a_data`, `b_data` 2×36; `a_valid`, `b_valid`; `x_ready`, `y_ready`; `mode_sigs` 8; `link_in` 2	`x_data`, `y_data` 2×36; `x_valid`, `y_valid`; `a_ready`, `b_ready`; `cmp` 2; `link_out` 2
`fifo`	`wrdata` 64; `wrreq`; `rdreq`	`rddata` 64; `full`; `empty`; `almost_full`
`cam`	`wkey`, `skey` 2×32; `waddr` 6; `we`	`match`; `mindex` 6
`smem`	`data` 32; `wcol` 4; `addr1` 6; `we1`	`out` 320

Table 5.4: RTL interfaces of new Flow FPGA hard blocks, showing a selection of the most important interface signals.

Block / mode	vs.	Soft	Wiring	Chan. w.
Butterfly DSP mode	FFT butterflies in soft logic	−67 %	−39 %	+15 %
cmul DSP mode	beamformer on 80 paired DSPs	0 %	−24 %	0 %
Stream memory (64×320)	original port geometry	0 %	+10 %	−35 %
Narrow write port	fabric staging register	−12 %	−23 %	0 %
Hardened FIFO tile	M20K + soft pointers	−35 %	−67 %	−37 %
Steering block	MSPM compactor tree in soft logic	−59 %	+3 %	−19 %
CAM	soft associative match (control fabric)	−63 %	−38 %	+12 %

Table 5.5: Benefit of each new Flow FPGA block in isolation.

be combined to form one large logical table. While the design of the hard block macro library has significant influence on the design of the blocks (such as the ability to automatically chain blocks), in this work this was only minimally done and all hard block usage is manually performed in the RTL.

Feature	Comparison	Wiring	Chan. width	Device
Stream-memory tile	Fabric with the stream-memory column replaced by ordinary memory blocks: the dot-product kernels' wide stream stores have no compatible site	unplaceable → routes		
Wide-port stream store	80-lane dot-product weight store as one wide-port memory vs. as 80 distributed narrow memory banks (the banked version needs a wider fabric to hold them)	+12 %	−48 %	−29 %

Table 5.6: Effects of Flow FPGA memory changes.

5.6 Results

5.6.1 Quantification of Individual Improvements

The Effects of Hard Blocks. The core reason for adding hard blocks was to remove soft logic and routing that would otherwise have been quite large, while at the same time being reusable across multiple kernels or in that particular class of kernels. The effects of adding each hard block are shown in the ablation study Table 5.5, which links each block to the kernel or property first responsible for its existence. As can be seen, by implementing only a small number of new hard blocks, significant improvements can be made to routing and soft logic.

The Effect of Wider Stream Memory. In comparison, the new wider memory actually increases the wiring required as in Table 5.6, but this is an artifact of the density of wires flowing out of this block. The real benefits are from saving energy through the consolidation of a large number of memory blocks into one.

The Effect of Nearby IO Access. Structurally, evaluating this involves measuring how structural properties like wiring, channel width, and soft logic change with

Circuit	I/O share of wiring	Extra wiring			Extra routing energy (both)
		far pads	bus	both	
Stream buffer	61 %	+18 %	+18 %	+49 %	+37 %
Beamformer stage	52 %	+10 %	+16 %	+25 %	+16 %
KAN classifier	22 %	+1 %	+7 %	+10 %	+19 %
Polyphase filter	9 %	+1 %	−2 %	−1 %	+5 %

Table 5.7: Impact of distance and buses on wiring to IO pads.

Kernel	Fabric columns	Device/str.	Soft/str.	Wiring/str.	Chan. w.
Channelizer FIR (conv2pt)	L^{SF}|M|D|D|L^{SFC}	−34 %	−31 %	−29 %	+10 %
Streaming FFT (butterfly mode)	L^{SF}|M|D|M|L^{SFC}	−27 %	−68 %	−30 %	+46 %
Beamformer (cmul mode)	L^{SF}|M|D|D|L^{SFC}| M|D|D|L^{SF}	−18 %	−19 %	−34 %	+64 %
Aggregation (CAM)	L^{SF}|M|D|D|L^{SFC}	−4 %	−63 %	−38 %	+12 %
MSPM compactor (steer2 + FIFO)	L^{SF}|M|D|D|L^{SFC}	−36 %	−75 %	−31 %	+41 %
KAN classifier (soft-only strip)	L^{F}|L^{F}|L^{F}|L^{F}|L^{F}	−28 %	−1 %	−18 %	+25 %
Mean		−25 %	−50 %	−30 %	+32 %

Table 5.8: Flow FPGA kernel design sweep results compared to a VTR/Koios baseline with Agilex-like provisioning.

distance from IO ports, in a Flow FPGA-like arrangement compared to a square FPGA with perimeter IO as in Table 5.7. It can be seen that for kernels that are low on wiring, as a kernel with a mostly-hard-block layout should be, the distance from the IO pads does add to the length of the wire and therefore the energy (captured in VTR 40nm [52]). This justifies the Flow FPGA aspect ratio. Notably, this experiment evaluates each change in isolation; therefore, the polyphase filter has not been fit into the new hard blocks, therefore, the IO share is a relatively small quantity in the overall wiring. In the version with hard blocks, the IO share would be larger.

Kernel	Device	Wiring	Chan. width
Convolution layer	+200 %	+73 %	+43 %
Reduction tree	+147 %	+57 %	+80 %
32-bit RISC CPU (OR1200)	+152 %	+39 %	−9 %
Blob detector	+144 %	+50 %	+33 %
Element-wise layer	+112 %	+53 %	+83 %
Softmax	+112 %	+13 %	−3 %
Sparse matrix–vector	+61 %	+21 %	+4 %
Systolic array (TPU-like)	−18 %	+105 %	+57 %

Table 5.9: Negative examples on Flow FPGA from Koios kernels.

5.6.2 Kernel-level Improvements

In this structural comparison that focuses on device tiles, soft logic, and wire length, it can be seen that Flow is able to implement the transport kernel suite with a large reduction in total number of tiles, as well as a strong decrease in soft logic and wiring as seen in Table 5.8. For inner product, the soft logic and wiring were already quite limited, as the majority of the kernel was already in hard blocks as shown in Figure 5.3. Therefore, the power improvements for this kernel will come from the better stream memory hard blocks instead. For DFT, aggregation, and MSPM, considerable soft logic reductions are made using the new hard blocks. A common trend in all is a large increase in channel width, reflecting the more concentrated bulk data movement across the Flow fabric. This simply controls the width of the routing tract, not the overall wiring, which uniformly decreases. However, as the model has only priced used wiring, it does not account for leakage and area from the wider routing tracts. Therefore, this is projected to significantly decrease power compared to the control.

As a negative control, other kernels from the Koios benchmark were also run on a Flow fabric, sized to fit the hard block count in Table 5.9. The majority of them

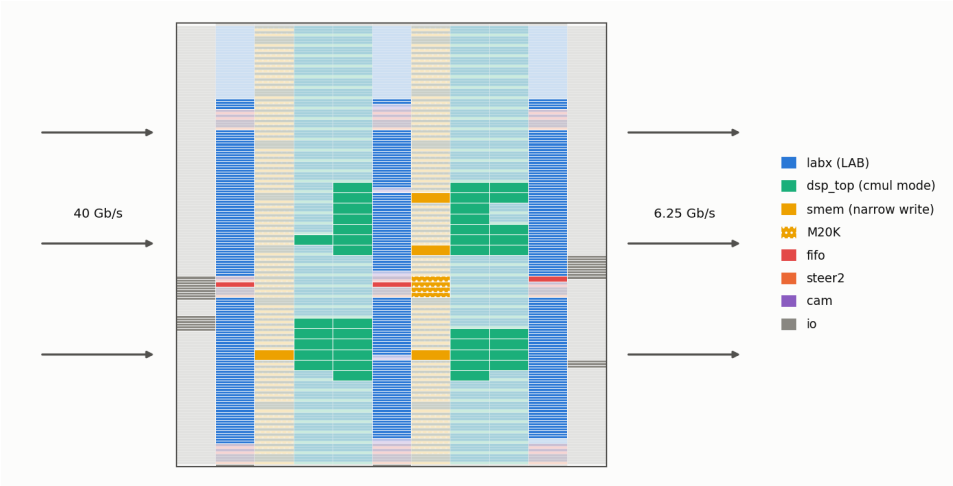


Figure 5.3: An example of placing the dotprod kernel, showing the new memory types and the increased hard block provisioning, as well as the much closer IO placement.

suffer due to the requirement of a fabric geometry and soft logic very different from what this fabric is provisioned for. A few interesting cases are mentioned and analyzed here. `systolic` is an interesting case because it is only compute blocks with no memory and minimal soft logic. Therefore, when fitting the Flow FPGA device, the compute block columns are easily obtained. However, the extra wiring comes from having to fold a 2D array into Flow FPGA's more 1D-like shape, as well as the block cascades being present along the long edge of the fabric does not allow direct chaining across. `convolution` fails because of soft logic provisioning, as for a Flow FPGA-style fabric to reach the same soft logic required it has to be much larger. `elementwise`, which one would have expected to do well on Flow FPGA as a simple feedforward computation, fails because the benchmark design is a memory-resident design with hardly any IO pad usage (only for control), and as such requires a large amount of RAM blocks for holding inputs. From these examples, the Flow FPGA fabric is clearly specialized towards its particular processing modality, while achiev-

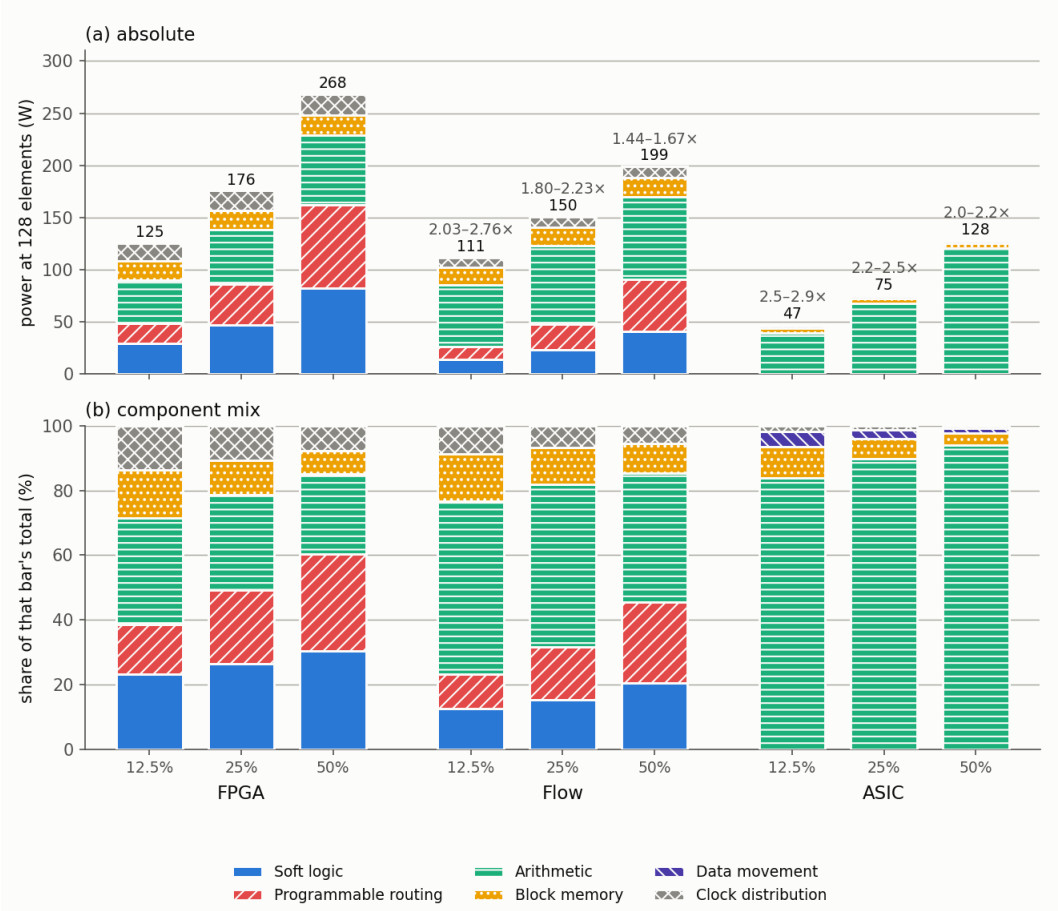


Figure 5.4: Flow compared to ASIC baseline.

ing better efficiency in the kernels it was meant for, validating the design choices as real tradeoffs. Additionally, these examples do not rule out instantiations of these algorithms in a data transport computing manner, as for example an element-wise layer would fit easily into the modality.

	Power (W)			Ratio to ASIC		Gap
Activity	FPGA	Flow FPGA	ASIC	FPGA	Flow FPGA	closed
12.5 %	125.4	102.7–120.1	43.5–50.5	2.48–2.88×	2.03–2.76×	6–30 %
25 %	175.6	141.2–159.7	71.5–78.5	2.24–2.46×	1.80–2.23×	15–35 %
50 %	268.5	189.1–208.4	124.5–131.5	2.04–2.16×	1.44–1.67×	42–58 %

Table 5.10: Flow FPGA energy results, showing that architectural changes close the gap between the commercial FPGA and the ASIC baseline.

5.7 Comparison with the Chapter 4 Baseline

A Note on Methodology. One cannot directly compare Flow power numbers with Quartus [75] or OpenROAD ASAP7 [92, 93] derived power numbers as they are on vastly different technologies, and moreover the commercial device has a large number of optimizations that are not available in the academic toolchain. To make the comparison feasible therefore, a simplifying assumption was made. It is assumed that the structural improvements in Flow over the comparable Agilex-like Koios-derived baseline would carry over to the real device, **as the changes are all broadly architectural in nature, such as new hard blocks and aspect ratio, and not dependent on specific fabric structure.** Following this, like-for-like improvements are projected onto the Quartus on Agilex results. For example, if the Flow FPGA model achieved a 50% reduction in soft logic over the baseline, the Quartus soft logic would be decreased by 50%. Then, power was estimated as before, using Power Analyzer and Power and Thermal Calculator breakdowns. Any new blocks were priced using the Chapter 4 ASIC flow as relevant. This is at best an approximation, but required due to the vastly different engineering and technology nodes. Additionally, this methodology still suffers from the disproportionately rising ASIC power from simulation at high toggle rates.

Comparisons. As seen in Table 5.10, architectural improvements are able to make progress in closing the gap between Agilex FPGA and the ASIC baseline, starting from 6–30% at 12.5% change to 42–58% at 50% change, with variance due to the same factors as in Chapter 4. This is also visually seen in Figure 5.4, as soft logic and routing are reduced compared to the base FPGA results, with a much larger fraction of Flow power coming from hard blocks. As a reminder, this is solely due to architectural effects from the new fabric composition and does not include any circuits or CAD level changes, which would further optimize the Flow device. The gap closure increases with toggle rate as soft logic and routing become disproportionately large contributors towards total power, rendering Flow FPGA's structural improvements that much more consequential. **This demonstrates the second claim of the thesis, that with domain-specific optimizations, the specialized FPGA can significantly close the gap to ASICs, making their engineering costs redundant.**

5.8 Conclusion

In this chapter, a new FPGA family called Flow was proposed, which is a result of domain-specific optimization of FPGA fabric organization for the data transport computing domain. In particular, the Flow FPGA is specialized towards the short bounded feedforward pipeline archetype of data transport computing. This is achieved through three specific optimizations: short IO-to-IO distance and good support for short pipelines; a concentrated focus on domain-specific hard blocks and lower soft logic for higher efficiency; and new memory and streaming data blocks such as wide RAM, FIFO, and steering primitives that express common streaming data movement operations inside the fabric. A toolchain was developed for the Flow

FPGA fabric using VTR, with a templatable design generated by a script. This allows for instantiating various instances of the Flow fabric with different hard block and soft logic columns, with the added feature of hard blocks optionally being distributed in the soft logic for when too few of them are required to warrant an entire column, but their effects important. On running designs sweeps on a set of data transport kernels drawn from the application domains, Flow achieved a mean -25% device tile count, -50% soft logic, and -30% wiring over an Agilex-like capture in Koios [99] baseline. On negative benchmark examples, kernels that did not fall into the data transport computing modality or any reason, such as IO harness to computation shape, performed significant worse on Flow FPGA than on the baseline fabric, showing a real tradeoff. When translated into power, Flow FPGA closed 6–30% (at 12.5% toggle) to 42–58% (at 50% toggle) of the gap between the Agilex FPGA ASIC baseline in the SOAP benchmark kernels. This was all done through architectural changes in FPGA fabric, without altering its programmability or flexibility benefits. This demonstrates the second claim of the thesis, that with domain-specific optimizations, the specialized FPGA can significantly close the gap to ASICs, making their engineering costs redundant. (with all the methodology caveats underscored above), without losing their base programmability and flexibility advantages. It is expected that with circuit and CAD level changes, such as lower swing drivers and a clock tree optimized for Flow-like usage, even more efficiency can be achieved.

Chapter 6

Physical and Logical Integration

6.1 Introduction

This is a short chapter exploring some of the system-level consequences of the Flow FPGA design. This chapter is more speculative than the three before, in that it provides a speculative look at the effects of interface choice, justified by literature survey, to answer what a system with an integrated Flow FPGA might look like. This is drawn from the observation that data transport kernels process a relatively large amount of IO for the amount of fabric resources they use up, even in a more optimized Flow FPGA design. This makes it essential to ensure that IO ingress, egress, and movement across the kernel are as cheap as possible, and as Flow kernels consume relatively little power, the question of the interface becomes important. Additionally, a few consequences for host-side design will also be explored.

6.2 Physical Integration

As mentioned in the data movement background in Chapter 2, there are two major types of interface.

Serial Interfaces. However, to get that data from the outside world to the fabric, it can arrive either over a narrow but fast serial interface, converted to the internal parallel wires via a SERDES (serializer-deserializer) or a wide parallel interface. To carry the required bandwidth, each wire lane in a serial interface must be clocked extremely high, and it requires an elaborate analog and digital endpoint to perform the conversion and to place and read the high-frequency signals. Due to the smaller number of wires that need to be kept in sync, serial interfaces are primarily used across large physical distances, where the transmission of a robust signal across meters or more is paramount. For example, Peripheral Component Interconnect Express or PCIe, which is the common standard for connecting CPUs and accelerators that are physically separated, operates over a multi-lane (up to 16-lane) serial link, requiring that number of serial transceivers at both ends [32]. Ethernet is also run over serial links, as network cables are driven by serial transceivers as well. The high frequency, relatively long supported distances, and the complex SERDES hardware make serial links relatively expensive from an energy standpoint. Therefore, this is only appropriate if the Flow FPGA device is physically separated from the system endpoint that it is supporting by a relatively large distance.

Parallel Interfaces. As packaging and integration technology have improved, there has been a lot of recent development in high-bandwidth parallel interfaces, driven by requirements such as High-Bandwidth Memory (HBM), which places a memory die right next to the computational die [82]. The wires are run over an

Technology	pJ/bit	Source	P @ 40 Gb/s	% of kernel
PCIe Gen5 SerDes PHY (incl. PLL, clocking; 32 Gb/s, 10 nm)	11.4	[32]	456 mW	29–61 %
PCIe Gen6 / CXL 3.0 transceiver (64 Gb/s, Intel 3)	4.6	[100]	184 mW	12–25 %
Ethernet-class 112 Gb/s PAM-4 PHY (5 nm)	4.63	[31]	185 mW	12–25 %
UCIe 1.0, standard package (spec. target; 100–130 µm pitch, ≤25 mm reach)	0.5	[101]	20 mW	1.3–2.7 %
HBM3 PHY (7.2 Gb/s/pin, Intel 4, EMIB)	0.5	[102]	20 mW	1.3–2.7 %

Table 6.1: Energy cost of carrying one lane's ingress (40 Gb/s) across interface technologies.

interposer, creating a very low power high bandwidth link that is ideally suited for close-device integration [102]. While Flow FPGA is not meant for memory traffic and as such does not need all the bandwidth of a HBM link, having an analogous wide parallel IO interface is beneficial. Additionally, the wires could possibly map more easily to the parallel input and output buses, making on-ramping and off-ramping from the fabric easier.

From Table 6.1, which measures the percentage of power burned by IO input to feed the inner product kernel, it is eminently visible that serial interfaces lag far behind near-device packaging and parallel interfaces. Therefore, for the IO-dominated Flow FPGA kernels, it almost requires close packaging with wide parallel interfaces to be efficient.

6.3 Logical Integration

The physical integration is a manufacturing-level decision that decides how Flow FPGA is to be packaged and connected to networks, endpoint devices, and other sources and sinks of data traffic. The logical interface instead deals with the world-view that logical devices employ when communicating with Flow FPGA. A key

Interface feature	Required structure or feature
Addressed memory access	memory controller; load/store units; request/response queues; MSHRs; ordering logic
Address translation	TLBs; page-table walker; fault-handling path; protection state
Shared memory	unified address map; atomics; fences
Cache coherence	directory or snoop filters; coherence interconnect; per-line state machines
Packetized transport (Ethernet-class)	MAC, PCS, FEC; framing/parsing; DMA with descriptor rings; packet buffers
Transaction protocol (PCIe-class)	tagged request tracking; completion reordering; credit-based flow control; replay and retry buffers
Ordering and multiplexing	arbiters; virtual channels; reorder buffers; per-flow state
Raw bitstream	wires; a FIFO; a valid/ready handshake

Table 6.2: What each logical-interface feature requires in hardware, contrasted with a raw bitstream. [1, 2]

consequence thereof is that Flow FPGA would require the correct architectural and micro-architectural features for any required logical interface features. As an example, if an endpoint device expresses a memory-like cache coherent interface with Flow FPGA, it requires the presence of memory addressing, address translation, shared memory semantics, and a cache coherence protocol running over the physical interface between the two devices [2, 103]. This further requires that the Flow FPGA FPGA implement such microarchitectural features as device TLBs, memory controller, fences, and coherence directories. A short accounting of common logical interface features is given in Table 6.2.

Each additional logical feature increases the complexity of the interface and the structures that need to be built, weakening the argument for Flow FPGA as a simple streaming device. Therefore, the most ideal method is a simple FIFO that transmits raw bits, failing which, a packetized Ethernet interface is probably ideal as it is the closest to a streaming transport. However, commercial devices today such

as CPUs and GPUs do not offer a raw bits or direct Ethernet connection, limiting the viability of Flow FPGA as an integrable device.

The above discussion assumes that the Flow FPGA is explicitly integrated as its own termination and endpoint, requiring the full suite of interface features. Another option is to integrate Flow FPGA transparently along the data wires of an existing (possibly complex) logical interface in a bump-in-the-wire fashion, which has precendent [16, 17]. With an appropriate "decoder", which could, for example, extract raw bits from a memory transaction or network packet while ignoring other protocol-specific bits, a Flow FPGA FPGA could operate on top of an existing protocol, as long as said protocol is amenable to its data being modified on the fly (with appropriate checksums etc.). However, this is highly restrictive in the kinds of problems expressible and solvable and could potentially require more complicated structures such as packet reassembly (which could become a hard block). In summary, while modern systems today readily support the types of wide parallel low power physical interface required by Flow FPGA, they on the whole do not support a simple streaming logical interface, limiting the integration potential without major changes to the endpoints. Some of these are discussed below, along with a discussion on programming.

6.4 Application Integration and Programming

From the data transport computing scenario, the overall goal is for the transport compute to live as part of a larger application. This means that on top of a physical interface and a low-level logical interface, it is also important to consider what an application-level interface might be, as at this layer, new capabilities are required that live on top of the physical and logical interfaces. For example, a key inter-

face operation is the ability to schedule and manage multiple data Flow FPGAs, and to virtualize the device among a number of applications [104]. This enables application-level features such as load balancing and isolation for security, which could be important in the right context, such as ensuring that a query processing engine is both fair in the sense that workloads do not starve, and secure in the sense that one query cannot interact with another. The Flow FPGA device as presented so far has not considered any application-level interface, and this would be unable to solve either of these issues. If such application-level features are desired, they could be implemented either on Flow FPGA itself, requiring a flexible data routing and management block before the fabric itself which would increase the required power. This is essentially integrating a hardware router. Otherwise, this flexibility could be handled on the endpoint device instead, with Flow FPGA still relegated to a simple streaming device. The tradeoff in between the two would depend on the precise logical interface and whether the endpoint device is able to marshal the data correctly.

Flow FPGA Kernel Programming. A major strength of Flow FPGA being a specialized FPGA instead of a generalized ASIC is that Flow FPGA kernels are already programmable through RTL input into FPGA toolchains without requiring the costly development of a bespoke ASIC toolchain. The VTR-based framework developed for Chapter 5 is implicitly a programming interface as well, and even today is able to place and route for a Flow FPGA fabric. Flow FPGA RTL would consist largely of instantiated hard blocks that are connected, controlled, and coordinated through RTL expressed in soft logic, with a few leftover or additional operators implemented in RTL. As the archetypical Flow FPGA kernel is a short, relatively simple feedforward pipeline, this RTL interface might be sufficient given the limited

scope of the programmable surface.

If a higher-level kernel programming abstraction is required, High-Level Synthesis (HLS) tools today can readily express feedforward pipelines with configurable initiation interval, replicates, and other key properties [76, 77]. This is exemplified by some of the SOAP kernels being themselves expressed as relatively simple oneAPI for FPGA HLS code [79]. The simplicity of the Flow FPGA design should make it easier for an HLS scheduler to map given code to Flow FPGA, as a non-domain-specific mapping would simply be too inefficient to use and should therefore error out. Therefore, it is feasible to imagine Flow FPGA RTL being generated by an HLS frontend instead of being written manually, allowing for the same benefits as HLS provides to general FPGA code, but to a greater extent given the more restricted problem space.

Flow FPGA System Programming. The final point is how Flow FPGA is programmed at a system level, in terms of scheduling data flow and kernel control. This largely depends on the desired application level interface. Compared to the complexity of compute-offload models such as CUDA, OpenCL, and SYCL, the usage modality of Flow FPGA only requires launching kernels and wiring up streams, as well as using the logical interface to decide where the resultant streaming data would go on the endpoint system [40, 79]. It may be enough to provide primitives to start and stop kernels on Flow FPGA locations, and connect named streams to and from them, as that is the only way for Flow FPGA kernels to interact with the outside world.

6.5 Conclusion

When it comes to integration at the physical, logical, and application level, Flow FPGA is an atypical class of device as it calls for a pure streaming interface and not the more commonly used memory-like interface features. At the physical level, IO costs should be minimized through integration over wide parallel interfaces. However, at the logical level, there is a severe mismatch between Flow FPGA streaming and for example GPU and CPU memory mapped IO, requiring the development of additional interface translation hardware either on the Flow FPGA fabric or on the endpoint device. Additionally, application-level concerns such as load balancing and isolation require complex logical interfaces as well that allow the user to freely remap where data streams go, making the design of such an interface a key follow-up question. The kernel and system programming for Flow FPGA, however, is relatively simple, given the constrained design space and the usage of FPGA fabric allowing existing tools to be re-used with relatively little difficulty.

Chapter 7

Conclusion & Future Work

7.1 The Big Picture

As stated in Chapter 1, at a very high level this dissertation is on the problem of computing on data in the communications domain. This is the inspiration behind the definition of Data Transport Compute offered in this work, and of the particular constraints chosen for the problem, namely that the schedule is externally decided (because the communication layer is a transport engine in service to its endpoints) and that the computation has bounded reuse and latency (because the external schedule imposes a particular structure, and a pipeline should not create a larger communication or buffering problem than the initial one). Having stationary operands is a practical constraint from the structure of many of these kernels. This is an attempt to carve a precise boundary within the computation-in-communication space, which other works have already addressed before, but not along this particular definition and flexibility.

The proposed solution is an FPGA, a device that is unusually well-suited to streaming problems just from its architecture, without requiring complex instruction-

level control. This was demonstrated using an engineering exercise of constructing a high-bandwidth radar system, where it was shown that modern FPGA fabric at high FMax and with its rich IO specification can both handle the computation and system integration sides, effectively pushing the computation into the communication layer for this problem and achieving the stated system-level goal in this big picture. Importantly, FPGAs achieve this without giving up bit-level soft logic flexibility that allows them to address a number of different kernels, as well as implement operators that may not be covered by hard block provisioning. An efficient programmable solution allows for customizable acceleration in the communications space, increasing the reach and viability of the solution and of expanding compute to this domain in the first place. In the resulting design study. The FPGA-based processing element grid system ingested up to 40.96Tbps of realtime data, and with 4x the GPU's efficiency in on the energy-dominant inner product kernels, cut the system power by 55.3% when added to the GPU baseline. Due to FPGA bandwidth reduction, the full exchange fit within a single commercial switching ASIC, with 23.1Tbps specified against 51.2Tbps available. The sweeps also showed that today's devices are transceiver-limited, with only 24% of its compute blocks exercised.

Further, as FPGAs are general-purpose devices, a very conservative gap was quantified between FPGAs and an ASIC baseline. ASIC models were created by constructing appropriate atoms and evaluating them through the open source ORFS toolchain. From this, it was shown that FPGA arithmetic, in its implementation, was already within range of the ASIC baseline, and what down efficiency is instead is data routing, soft logic, and data access within the fabric that are not optimized for this domain. The kernel gap was measured at 2.0x-2.9x across activity factors, with soft logic and routing accounting for 60% of FPGA power.

In response to this, a more purpose-built fabric called the Flow FPGA was proposed, constructed out of the visual image of vertically stacking parallel IO-bound pipelines, and the on-fabric hardware that would be required for this endeavor. The Flow FPGA fabric is shown to be a better fit for the problem domain than stock fabric, achieving a geomean of -25% device tiles, -50% soft logic and -30% wiring against an Agilex-geometry control baseline, bringing the FPGA-to-ASIC ratio down to 1.44-1.67x., although using a pen-and-paper power model where components are priced as they are in the Altera Agilex power capture, making it a rough estimate at best. An interesting property of Flow FPGA is the ability to host a small number of high-value blocks in the fabric, like CAM blocks which allowed the aggregation kernel to place and route on Flow FPGA as it absorbed a significant amount of soft logic. Additionally, the fabric is a parametrized template, allowing for custom design stidues based on the desired kernels.

Finally, a light evaluation and discussion was provided of the system-level integration options for a device such as the Flow FPGA, as integration and data movement energy into and across the fabric can be a sizable fraction of the power. It was shown, for example, that a serial link at 40Gb/s costs 29–61% of the dot product kernel's computation power, making close packaging with wide parallel IO (only 1.3–2.7%) obligatory. It was shown that for a device like Flow FPGA, a very simple streaming raw bits interface was enough, and did not require the deep complex memory-like paths of other accelerator interfaces, or the long distance of existing integration technology like PCIe. A short qualitative argument was then made for the integration of Flow FPGA close to devices, and its programming and integration into application systems, noting that it was an open problem of where to place the complexity of application-level management.

In short, this dissertation addresses the two thesis claims that FPGAs already address the problem of data transport computing more effectively than a traditional architecture like a GPU (55.3% power decrease from FPGA addition, diverse serial IO), and that with domain-specific optimizations they can close the gap to ASIC (up to 42–58% depending on toggle rate), rendering the latter less useful in projects where the engineering costs of the ASICs are higher than the FPGA-ASIC gap.

It therefore represents an attempt at the explicit qualification of a common type of computing challenge within the computing-in-communication domain, and demonstrates an early line of inquiry into more successfully solving computational problems in this domain.

7.2 Future Work

As the above states, the overall mission of the pursued line of work was to qualify and address the problem of computing on streaming data that is in transit in a computing system, for increased efficiency in the problems where this domain is a significant feature, such as streaming signal processing, network functions, and data analytics. This thesis proposed an early approach towards demonstrating a line of inquiry into this large problem space. The questions raised by this work therefore directly follow in terms of four major questions that arise for the systems-level challenge of solving this problem domain.

1. **What is the right computational substrate?** This work provided an early look at an optimized FPGA fabric. Follow-up work could model this device in greater detail, including new types of blocks, circuit and CAD level changes, new interfaces, and an expanded suite of kernels. Also valuable would be non-FPGA fabrics, or fabrics inspired by FPGAs but perhaps with coarser

structures if a particular computational domain requires it.

2. **What is the right system interface?** This includes exploring physical and logical interface, both in Flow-type devices as well as enabling the other side on network ports and other existing architectures. In particular, this includes the impedance mismatch problem of communicating with a device that has a memory-like interface. New packaging and physical integration technologies such as 3D stacking could also be explored. Another pertinent issue is what data mover structures need to be instantiated to make this integration happen.

3. **What is the right programming abstraction?** This new domain, with its streaming focus, being an FPGA, being very close to IO, and being closely integrated with the rest of the system raises the problem of how these devices would be programmed, a question not addressed in this dissertation. This includes both fabric programming for the actual compute, as well as the "full-system" programming of the integration of these devices, scheduling data flows across the fabric, etc.

4. **What is the right problem?** Many problems have hitherto only been solved in the traditional abstraction of completely separated communication, memory, and compute. In the hypothetical scenario that a Flow-like system were to be constructed, what problems and applications would be able to take advantage of this new computational regime, and can existing application be modified to take advantage of this, if certain components thereof can be cast into a computation-in-communication light?

As can be seen, a lot of problems in this line of questioning remain unsolved, or addressed ad-hoc at best, representing an interesting line of further inquiry.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Cambridge, MA, USA: Morgan Kaufmann, 6th ed., 2017. (document), 1.1, 1.2, 5.2, 6.2

[2] NVIDIA Corporation, "NVIDIA Grace Hopper superchip architecture whitepaper." https://resources.nvidia.com/en-us-grace-cpu/nvidia-grace-hopper, 2023. Accessed: 2026-08-18. (document), 1.3, 1, 6.2, 6.3

[3] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, p. 20–24, Mar. 1995. 1.1

[4] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google workloads for consumer devices: Mitigating data movement bottlenecks," *SIGPLAN Not.*, vol. 53, p. 316–331, Mar. 2018. 1.1

[5] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014. 1.1, ??, ??, ??, ??, ??, ??

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, p. 365–376, June 2011. 1.1, 1.4

[7] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. 1.1, 1.4

[8] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, p. 48–57, June 2020. 1.1, 1.1, 4.4, 4.2.2

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, p. 65–76, Apr. 2009. 1.1

[10] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Processing data where it makes sense: Enabling in-memory computation," *Microprocessors and Microsystems*, vol. 67, pp. 28–41, 2019. 1.1

[11] L. Woods, Z. István, and G. Alonso, "Ibex: an intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, vol. 7, p. 963–974, July 2014. 1.1, 2.4.3

[12] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *Computer*, vol. 40, no. 12, pp. 33–37, 2007. 1.1

[13] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin, J. Kim, O. Seongil, A. Iyer, D. Wang, K. Sohn, and N. S. Kim, "Hardware architecture and software stack for pim based on commercial dram

technology : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 43–56, 2021. 1.1

[14] J. Gómez-Luna, I. E. Hajj, I. Fernandez, C. Giannoula, G. F. Oliveira, and O. Mutlu, "Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system," *IEEE Access*, vol. 10, pp. 52565–52608, 2022. 1.1

[15] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, HotNets '17, (New York, NY, USA), p. 150–156, Association for Computing Machinery, 2017. 1.1

[16] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, 2016. 1.1, 1.4, 2.4.1, 6.3

[17] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg, "Azure accelerated networking: SmartNICs in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Imple-*

mentation (NSDI 18), (Renton, WA), pp. 51–66, USENIX Association, Apr. 2018. 1.1, 2.4.1, 6.3

[18] Z. F. Eryilmaz, A. Kakaraparthy, J. M. Patel, R. Sen, and K. Park, "Fpga for aggregate processing: The good, the bad, and the ugly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1044–1055, 2021. 1.1, 1.4, 1.5

[19] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *Proc. VLDB Endow.*, vol. 2, p. 910–921, Aug. 2009. 1.1, 2.4.3

[20] F. Harris, C. Dick, and M. Rice, "Digital receivers and transmitters using polyphase filter banks for wireless communications," *IEEE Transactions on Microwave Theory and Techniques*, vol. 51, no. 4, pp. 1395–1412, 2003. 1.1, 3.2.1

[21] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," *SIGARCH Comput. Archit. News*, vol. 38, p. 37–47, June 2010. 1.1

[22] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "Asic clouds: specializing the datacenter," *SIGARCH Comput. Archit. News*, vol. 44, p. 178–190, June 2016. 1.1, 1.4

[23] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, pp. 171–210, June 2002. 1.1

[24] A. Boutros and V. Betz, "Fpga architecture: Principles and progression,"

IEEE Circuits and Systems Magazine, vol. 21, no. 2, pp. 4–29, 2021. 1.1, 1.2, 1.2, 1.4, 2.1

[25] A. Boutros, E. Nurvitadhi, R. Ma, S. Gribok, Z. Zhao, J. C. Hoe, V. Betz, and M. Langhammer, "Beyond peak performance: Comparing the real performance of ai-optimized fpgas and gpus," in *2020 International Conference on Field-Programmable Technology (ICFPT)*, pp. 10–19, 2020. 1.1

[26] S. M. Trimberger, "Three ages of fpgas: A retrospective on the first thirty years of fpga technology," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 318–331, 2015. 1.1, 2.1

[27] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007. 1.1, 1.4, 1.4, 4, 4.1.1, 4.3, 4.5

[28] A. Arora, M. Ghosh, S. Mehta, V. Betz, and L. K. John, "Tensor slices: Fpga building blocks for the deep learning era," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, Aug. 2022. 1.1

[29] L. Rizzo, "netmap: A novel framework for fast packet I/O," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, (Boston, MA), pp. 101–112, USENIX Association, June 2012. 1.2

[30] DPDK Project, "DPDK: Data plane development kit." https://www.dpdk.org, 2026. Accessed: 2026-08-18. 1.2, 1.2

[31] H. Park, M. Abdullatif, E. Chen, A. Elmallah, Q. Nehal, M. Gandara, T.-B. Liu, A. Khashaba, J. Lee, C.-Y. Kuan, D. Hamachandran, R.-B. Sun, A. Atharav, Y. Chun, M. Zhang, D.-F. Weng, C.-H. Tsai, C.-H. Chang, C.-S.

Peng, S.-T. Hsu, and T. Ali, "A 4.63pj/b 112gb/s dsp-based pam-4 transceiver for a large-scale switch in 5nm finfet," in *2023 IEEE International Solid-State Circuits Conference (ISSCC)*, pp. 5–7, 2023. ??, 2.3.1, ??

[32] M. Bichan, C. Ting, B. Zand, J. Wang, R. Shulyzki, J. Guthrie, K. Tyshchenko, J. Zhao, A. Parsafar, E. Liu, A. Vatankhahghadim, S. Sharifian, A. Tyshchenko, M. De Vita, S. Rubab, S. Iyer, F. Spagna, and N. Dolev, "A 32gb/s nrz 37db serdes in 10nm cmos to support pci express gen 5 protocol," in *2020 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–4, 2020. ??, 6.2, ??

[33] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: energy-efficient dram for extreme bandwidth systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 41–54, Association for Computing Machinery, 2017. ??

[34] L. Shang, A. S. Kaviani, and K. Bathala, "Dynamic power consumption in virtex™-ii fpga family," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-Programmable Gate Arrays*, FPGA '02, (New York, NY, USA), p. 157–164, Association for Computing Machinery, 2002. ??

[35] Intel Corporation, "Intel data direct I/O technology (Intel DDIO): A primer." Technical Brief, Revision 1.0, https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/data-direct-i-o-technology-brief.pdf, Feb. 2012. Accessed: 2026-08-18. 1.2

[36] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A

unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008. 1.3

[37] J. Choquette, "Nvidia hopper h100 gpu: Scaling performance," *IEEE Micro*, vol. 43, no. 3, pp. 9–17, 2023. 1.3

[38] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," vol. 45, p. 1–12, June 2017. 1.3

[39] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–14, 2021. 1.3

[40] NVIDIA Corporation, "CUDA C++ programming guide." https://docs.

nvidia.com/cuda/cuda-c-programming-guide/, 2026. Accessed: 2026-08-18. 1.3, 6.4

[41] NVIDIA Corporation, "GPUDirect RDMA documentation." https://docs.nvidia.com/cuda/gpudirect-rdma/, 2026. Accessed: 2026-08-18. 1.3, 1

[42] NVIDIA Corporation, "NVIDIA BlueField-3 DPU datasheet." https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2026. Accessed: 2026-08-18. 1.3

[43] Intel Corporation, "Intel infrastructure processing unit (Intel IPU)." https://www.intel.com/content/www/us/en/products/details/network-io/ipu.html, 2026. Accessed: 2026-08-18. 1.3

[44] Silicom Ltd., "FPGA SmartNIC N6010 product brief." https://www.silicom-usa.com, 2026. Accessed: 2026-08-18. 1.3

[45] A. Agrawal and C. Kim, "Intel tofino2 – a 12.9tbps p4-programmable ethernet switch," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, pp. 1–32, 2020. 1.3

[46] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. R. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, "Think fast: a tensor streaming processor (tsp) for accelerating deep learning workloads," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, p. 145–158, IEEE Press, 2020. 1.3, ??

[47] R. Prabhakar, J. Zhou, D. Gandhi, Y. Choi, M. Khayatzadeh, K. Kim, U. Durairajan, J. Park, S. Sarkar, and J. L. Shin, "16.4: Sambanova sn40l: A 5nm 2.5d dataflow accelerator with three memory tiers for trillion parameter ai," in *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, vol. 68, pp. 288–290, 2025. 1.3, **??**, 5.4

[48] J. Vasiljevic and D. Capalija, "Blackhole & TT-Metalium: The standalone AI computer and its programming model," in *IEEE Hot Chips 36 Symposium (HCS)*, 2024. 1.3, **??**

[49] S. Lie, "Cerebras architecture deep dive: First look inside the hardware/software co-design for deep learning," *IEEE Micro*, vol. 43, p. 18–30, May 2023. 1.3, **??**, 5.4

[50] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 111–119, 2020. 1.4

[51] S. Naffziger, N. Beck, T. Burd, K. Lepak, G. H. Loh, M. Subramony, and S. White, "Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, p. 57–70, IEEE Press, 2021. 1.4

[52] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. P. Walker, H. Zeng, P. Patros, J. Luu, K. B. Kent, and V. Betz, "Vtr 8: High-performance cad and customizable fpga

architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, June 2020. 1.4, 3, 1.7, 5.2, 5.5.1, 5.6.1

[53] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Gaide, and Y. Arbel, "Network-on-chip programmable platform in versaltm acap architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), p. 212–221, Association for Computing Machinery, 2019. 1.4, 5.4

[54] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, (New York, NY, USA), p. 84–93, Association for Computing Machinery, 2019. 1.4

[55] Altera Corporation, "Agilex™ 7 FPGA M-Series product overview." https://www.altera.com/products/fpga/agilex/agilex-7/m-series, 2026. Accessed: 2026-08-18. 1.4, 5.4

[56] Achronix Semiconductor Corporation, "Speedster7t FPGAs with 2D network-on-chip." https://www.achronix.com/product/speedster7t-fpgas, 2026. Accessed: 2026-08-18. 1.4, 5.4

[57] A. Boutros, E. Nurvitadhi, and V. Betz, "Architecture and application co-design for beyond-fpga reconfigurable acceleration devices," *IEEE Access*, vol. 10, pp. 95067–95082, 2022. 1.4

[58] M. S. Abdelfattah and V. Betz, "Networks-on-chip for fpgas: Hard, soft or mixed?," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, Sept. 2014. 1.4

[59] M. S. Abdelfattah and V. Betz, "Power analysis of embedded nocs on fpgas and comparison with custom buses," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 1, pp. 165–177, 2016. 1.4

[60] AMD, Inc., "Versal adaptive SoC AI engine architecture manual (AM009)." v1.4, https://docs.amd.com/r/en-US/am009-versal-ai-engine, 2026. Accessed: 2026-08-18. 1.4

[61] J. Capon, "High-resolution frequency-wavenumber spectrum analysis," *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, 1969. 1.5, 3.2.1

[62] J. Gray, A. Bosworth, A. Lyaman, and H. Pirahesh, "Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals," in *Proceedings of the Twelfth International Conference on Data Engineering*, pp. 152–159, 1996. 1.5

[63] Transaction Processing Performance Council, "TPC benchmark H (TPC-H) standard specification." Revision 3.0, https://www.tpc.org/tpch/, 2021. Accessed: 2026-08-18. 1.5

[64] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), p. 215–226, Association for Computing Machinery, 2016. 1.5

[65] Z. István, G. Alonso, M. Blott, and K. Vissers, "A hash table for line-rate data processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, Mar. 2015. 1.5, 2.4.3

[66] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1083–1100, USENIX Association, Nov. 2020. 1.5

[67] Z. Zhao, J. Melber, S. Sahay, S. Obla, E. Nurvitadhi, and J. C. Hoe, "Exploiting the common case when accelerating input-dependent stream processing by fpga," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1343–1355, 2023. 1.5

[68] D. Hoang, A. Gupta, and P. C. Harris, "Kanelé: Kolmogorov–arnold networks for efficient lut-based evaluation," in *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '26, (New York, NY, USA), p. 44–55, Association for Computing Machinery, 2026. 1.5

[69] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, p. P07027, jul 2018. 1.5

[70] D. Lewis, G. Chiu, J. Chromczak, D. Galloway, B. Gamsa, V. Manohararajah, I. Milton, T. Vanderhoek, and J. Van Dyken, "The stratix™ 10 highly pipelined fpga architecture," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, (New York, NY, USA), p. 159–168, Association for Computing Machinery, 2016. 2.1, 5.3.3

[71] M. Langhammer, G. Baeckler, and K. Bozman, "A 950 mhz simt soft proces-

sor," in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1228–1235, 2025. 2.1, 4.2

[72] M. Langhammer, E. Nurvitadhi, S. Gribok, and B. Pasca, "Stratix 10 nx architecture," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, Aug. 2022. 2.1, 2.4.4

[73] Intel Corporation, "Agilex™ 7 FPGAs and SoCs F-Series Product Table," September 2025. Accessed: 2025-12-02. 2.1, 2.3.2, 5.2, 5.3, 5.5.1

[74] J. Boo, Y. Chung, E. Baek, S. Na, C. Kim, and J. Kim, "F4t: A fast and flexible fpga-based full-stack tcp acceleration framework," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, (New York, NY, USA), Association for Computing Machinery, 2023. 2.1

[75] Intel Corporation, *3.1. Design Flow.* Intel Corporation, Oct. 2021. In: Quartus® Prime Standard Edition User Guide: Design Compilation. 2.2, 4.2.1, 5.7

[76] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 30, p. 473–491, Apr. 2011. 2.2, 6.4

[77] Advanced Micro Devices, Inc., "AMD Vivado™ High-Level Design," 2025. Part of the Vivado™ Software product page. 2.2, 6.4

[78] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2021. 2.2

[79] Intel Corporation, *Compilation Flow Overview*. Intel Corporation, Oct. 2024. In: Intel® oneAPI Programming Guide. 2.2, 6.4, 6.4

[80] NVIDIA Corporation, "NVIDIA Spectrum-4 SN5000 switch systems hardware user manual: Specifications." https://networking-docs.nvidia.com/sn5000hw/specifications, 2026. Accessed: 2026-08-18. 2.3.1

[81] Broadcom Inc., "Broadcom ships Tomahawk 6: World's first 102.4 Tbps switch." Press release, https://www.broadcom.com/company/news/product-releases/64031, 2025. Accessed: 2026-08-18. 2.3.1, 3.3.2

[82] JEDEC Solid State Technology Association, "High bandwidth memory (HBM4) DRAM (JESD270-4)." https://www.jedec.org/standards-documents/docs/jesd270-4a, Apr. 2025. Accessed: 2026-08-18. 2.3.1, 6.2

[83] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," *Commun. ACM*, vol. 59, p. 114–122, Oct. 2016. 2.4.1

[84] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2016. 2.4.2

[85] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Bianchi, "Flowblaze: stateful packet processing in hardware," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (USA), p. 531–547, USENIX Association, 2019. 2.4.2

[86] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco, "hxdp: Efficient software packet processing on fpga nics," *Commun. ACM*, vol. 65, p. 92–100, July 2022. 2.4.2

[87] B. Van Veen and K. Buckley, "Beamforming: a versatile approach to spatial filtering," *IEEE ASSP Magazine*, vol. 5, no. 2, pp. 4–24, 1988. 3.2.1

[88] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020. 3.2.2

[89] NVIDIA Corporation, "NVIDIA DGX B200 datasheet." https://resources.nvidia.com/en-us-dgx-systems/dgx-b200-datasheet, 2026. Accessed: 2026-08-18. 3.3.2

[90] J. D. C. Little, "A proof for the queuing formula: L = w," *Oper. Res.*, vol. 9, p. 383–387, June 1961. 3.3.3, 5.5

[91] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop*

on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS), 2017. 3.3.3

[92] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem, G. Pradipta, S. Reda, M. Saligane, S. S. Sapatnekar, C. Sechen, M. Shalan, W. Swartz, L. Wang, Z. Wang, M. Woo, and B. Xu, "Toward an open-source digital flow: First learnings from the openroad project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, (New York, NY, USA), Association for Computing Machinery, 2019. 4, 4.2.2, 5.7

[93] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016. 4, 5.7

[94] C. Wolf, "Yosys open SYnthesis suite." https://yosyshq.net/yosys/. Accessed: 2026-08-18. 4.1.1, 4.2.2

[95] M. Freitas, N. Guimarães, R. Maria, F. Costa, G. Milani, B. Sanches, and W. V. Noije, "Using open source eda tools in asics for hep: A mixed comparison," *Journal of Instrumentation*, vol. 20, p. P12022, dec 2025. 4.1.1

[96] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Trans. Archit. Code Optim.*, vol. 14, June 2017. 4.4, 4.2.2

[97] A. Stillmaker and B. Baas, "Scaling equations for the accurate prediction of cmos device performance from 180nm to 7nm," *Integration*, vol. 58, pp. 74–81, 2017. 4.2.2

[98] Intel Corporation, "Intel Stratix 10 GX 10M FPGA: Emulation and prototyping." https://www.intel.com/content/www/us/en/design/test-and-validate/programmable/emulation-and-prototyping.html, 2026. Accessed: 2026-08-18. 5.2

[99] A. Arora, A. Boutros, S. A. Damghani, K. Mathur, V. Mohanty, T. Anand, M. A. Elgammal, K. B. Kent, V. Betz, and L. K. John, "Koios 2.0: Open-source deep learning benchmarks for fpga architecture and cad research," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 42, p. 3895–3909, Nov. 2023. 5.3.3, 5.5.1, 5.8

[100] D.-M. Choi, Y. Dong, R. Nicholson, F. Liu, W. Jia, V. Levin, M. He, S. Pradhan, J. Du, M. De Vita, A. Tran, R. Navid, S. Iyer, and R. Song, "A 4.6pj/b 64gb/s transceiver enabling pcie 6.0 and cxl 3.0 in intel 3 cmos technology," in *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*, pp. 1–2, 2024. ??

[101] UCIe Consortium, "Universal chiplet interconnect express (UCIe) specification, revision 1.0." https://www.uciexpress.org, 2022. Accessed: 2026-08-18. ??

[102] A. H. Mehta, J. S. Gaggatur, M. M. Rashid, S. Dakshinamurthy, A. K. L. Srinivasamurthy, A. K. Goyal, S. Manam, H. Gupta, S. Sukumar, V. K. Mishra, K. N. S, S. Nekkanti, S. Mitra, P. K. Jadhav, M. ChandraShekar, D. Humayun, M. C. Rifani, J. Xie, and A. P. Collins, "A 0.5pj/bit 7.2gbps hbm3 phy on intel4 with emib packaging and unmatched receiver architecture on phy side with per bit deskew correction," in *2025 38th International Conference on*

VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID), pp. 487–492, 2025. ??, 6.2

[103] S.-P. Yang, M. Kim, S. Nam, J. Park, J. yong Choi, E. H. Nam, E. Lee, S. Lee, and B. S. Kim, "Overcoming the memory wall with CXL-Enabled SSDs," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, (Boston, MA), pp. 601–617, USENIX Association, July 2023. 6.3

[104] A. Vaishnav, K. D. Pham, and D. Koch, "A survey on fpga virtualization," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 131–1317, 2018. 6.4

[105] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (New York, NY, USA), p. 365–376, Association for Computing Machinery, 2011.