2018

Cooperative high-performance computing with FPGAs - matrix multiply case-study

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Thesis

COOPERATIVE HIGH-PERFORMANCE COMPUTING WITH FPGAS - MATRIX MULTIPLY CASE-STUDY

by

ROBERT P. MUNAFO

A.B., Dartmouth College, 1986

Submitted in partial fulfillment of the

requirements for the degree of

Master of Science

2018

Approved by

First Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

Second Reader

Michel A. Kinsy, PhD
Assistant Professor of Electrical and Computer Engineering

Third Reader

Tali Moreshet, PhD
Senior Lecturer & Research Assistant Professor of Electrical and Computer Engineering

A problem is only a problem when viewed as a problem.
All change is hard at first, messy in the middle
and gorgeous at the end.

— Robin Sharma

Acknowledgments

I wish to acknowledge my advisor Prof. Martin Herbordt for the many meetings to discuss and clarify ideas, and for extensive patience. In addition I acknowledge all of my committee members for poking and prodding in new directions, and for imploring passionately for clarity and direction in my statements.

The other members of the CAAD lab (both present and past, the latter notably including Jaiyi Sheng Ph.D.) have answered many questions small and large and helped explain many bits of engineering design and methods that have made their way into this work.

It also goes nearly without saying that I have friends and family, without whose help and support I would not have gotten this far. Thank you, particularly to the friends and colleagues who encouraged me to go back to school in the first place.

— Robert Munafo

COOPERATIVE HIGH-PERFORMANCE COMPUTING WITH FPGAS - MATRIX MULTIPLY CASE-STUDY

ROBERT P. MUNAFO

ABSTRACT

In high-performance computing, there is great opportunity for systems that use FPGAs to handle communication while also performing computation on data in transit in an "altruistic" manner–that is, using resources for computation that might otherwise be used for communication, and in a way that improves overall system performance and efficiency. We provide a specific definition of **Computing in the Network** that captures this opportunity. We then outline some overall requirements and guidelines for cooperative computing that include this ability, and make suggestions for specific computing capabilities to be added to the networking hardware in a system. We then explore some algorithms running on a network so equipped for a few specific computing tasks: dense matrix multiplication, sparse matrix transposition and sparse matrix multiplication. In the first instance we give limits of problem size and estimates of performance that should be attainable with present-day FPGA hardware.

Contents

List of Tables

List of Figures

List of Abbreviations

AC	Altruistic Computing
ALM	Adaptive Logic Module
API	Application Programming Interface
BRAM	Block RAM
CAAD	Computer Architecture and Automated Design
CiN	Compute (or Computing) in the Network
CPU	Central Processing Unit
CUDA	*(Nvidia trademark, not an abbreviation)*
DDR	Double Data Rate
DPI	Direct and Programmable Interconnects
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First-In, First-Out
FPGA	Field-Programmable Gate Array
GB	Giga (10^9) Byte
GFLOP	Giga (10^9) FLoating-point OPerations
GPU	Graphics Processing Unit
HPC	High-Performance Computing
IC	Integrated Circuit
IP	Intellectual Property (module in FPGA design)
MGT	Multi-Gigabit Transceiver
MIPS	Million(s of) Instructions Per Second
MMM	Matrix-Matrix Multiply
MPI	Message-Passing Interface
NIC	Network Interface Controller
PCIe	Peripheral Component Interconnect Express
RAM	Random-Access Memory
SIMD	Single-Instruction, Multiple Data
SUMMA	Scalable Universal Matrix Multiply Algorithm
TDP	Thermal Design Power
TFLOP	Tera (10^{12}) FLoating-point OPerations
TOR	Top-Of-Rack (network switch)
TPU	Tensor Processing Unit

Chapter 1

Introduction

High Performance Computing (HPC) is the domain of computing applications that are computationally intensive, including the simulation and modeling of physical systems. HPC applications provide results that cannot be obtained through physical laboratory experiments or direct measurements. As such, HPC provides essential benefits to society, including the enabling of a vast array of scientific research (Pres., 2005); also, HPC is an indispensable tool of engineering (NSF, 2006).

As described by (Herbordt, 2018), four “great problems” for designing computer systems for HPC are as follows: (i) computational efficiency (getting the most GFLOPs out of available chip area), (ii) minimizing power usage, (iii) maintaining performance with portability, and (iv) handling the *communication bottleneck*, i.e., the increasing need for communication that accompanies ever more compute-intensive operations on ever growing datasets.

This last “great” problem is of particular interest to us in this thesis, and, as we will argue, it requires new architectures and design techniques.

Much research has addressed a similar problem regarding the latency of memory access. That work falls under the umbrella of *Compute in Memory*, and its main approach is to put computing capability closer to memory cells, in particular, within the memory chip itself. Examples include EXECUBE (Kogge, 1994), IRAM (Patterson et al., 1997), and Micron’s Hybrid Memory Cube (Pawlowski, 2011), (Gokhale et al., 2015).

Similarly, we propose to address the communication bottleneck with techniques that add computation capability to the devices that perform data communication; we refer to such capability as *Compute in the Network* (CiN). The approach we take is to provide ways for application's calculations to be combined with communication operations within a custom hardware design (specifically, with an FPGA). Properly applied, these techniques can address the communication bottleneck by providing dramatically reduced latency. For a more detailed introduction see (Herbordt, 2018).

As IBM BlueGene designer Paul Coteus is credited with saying (Herbordt, 2018), "For future computer systems to continue performance improvements, we need [to] compute everywhere, including in [the network] interfaces and [in the] network".

1.1 Outline

In the chapter 2 we will expound our main Thesis, including key definitions, taxonomies of system design and CiN capabilities, and our rationale for such. We outline proposed development tools for those who wish to add specific CiN capabilities to a system.

Chapter 3 describes some related prior work and many details of modern FPGAs that set constraints on what can be done in our proposed CiN model.

In chapter 4 we perform a thorough case study of CiN applied to dense matrix-matrix multiply. We give brief outlines of two other case studies (sparse matrix transpose and sparse matrix multiply).

We finish with a summary, implications of anticipated new technology, and suggestions for deeper inquiry into the present results and related CiN areas.

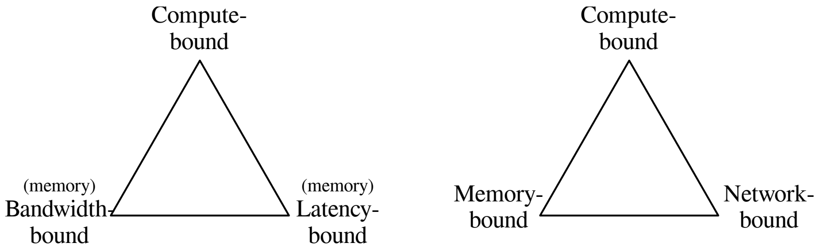


Figure 2·1: Commonly-cited design trade-off corners in the single node (left) and in multi-node systems (right). See text for details.

Chapter 2

Our Thesis, and Approach to Computing in the Network

Computer Science and Engineering (CSE) designers must make many design decisions, that are of varying but ever-finite longevity. As old decisions yield to new, the change is often driven by a need to specialize and/or consolidate, or by a need to generalize and/or broaden, the components and the capabilities of each component or subsystem and the scope of its intended application.

These decisions manifest themselves particularly strongly on the integrated circuit (IC)–by which we will implicitly include single-package multi-chip hybrids–because of the large costs (both of time and energy) of crossing the package boundary. For example, during the years that memory IC access times could keep up with CPU cycle times, all memory was external to the CPU IC. In the mid-1990s, the CPU cycle time became too fast for comparably-priced memory ICs to keep up, and memory (in the form of L1 cache, later L2 and L3) was added to the CPU. Similarly, communication ICs have almost entirely comprised communication functions such as switching, (de)coding, (de)modulation, except in some notable recent examples to be mentioned later. Likewise, memory ICs consist almost entirely of storage with some notable "Compute in Memory" exceptions we mentioned in the Introduction.

Around 2004-2006, Dennard scaling ended but Moore's law continued, making power an ever-growing constraint on transistor utilization and overall chip design. Thermal concerns make it increasingly difficult to concentrate more and more computing into an IC, and make it increasingly difficult to transfer data on- and off-chip at proportionately growing rates. These issues almost inevitably lead to a generalization and decentralization of functions, and a rise of the multifunction IC. Such multifunction ICs should be able to store data and also to compute (transform) data, and of course they need to incorporate communication functions; and all of these capabilities should be full-fledged rivals of the specialized ICs of the past.

This inevitable progression to multifunction ICs strongly suggests a mandate to capitalize on any opportunity to *compute on data in transit*, that is, to employ the communication network of an HPC system to facilitate more efficient execution of HPC applications. We give our Thesis, in the following mandate:

Perform each computation as soon as its inputs are in the same place at the same time; and maximize opportunities to do so by arranging for the utilization of the communication network, not just for relaying data, but also for operating upon it.

This is Computing in the Network (CiN).

2.1 Definitions and Classifications

2.1.1 The Domain of Interest

Our mandate to facilitate Compute in the Network applies particularly strongly to systems built for HPC applications. These are constructed as clusters of computing nodes that are equipped to work together running a single program. The alternative model, clusters and clouds that efficiently run applications in batch mode independently on multiple nodes, while important, does not depend as significantly on communication performance and so is much less relevant to CiN. In HPC systems, the nodes each have memory, computing capability (in a CPU possibly with other ICs such as a GPU), and an interface to a system-wide communication network. This interface could be an IC or set of ICs; we will refer to this interface as the node's Network Interface Controller (NIC). The NIC can initiate and receive data communications, but full end-to-end delivery of messages in non-trivial systems requires switching (or *routing*) of data at intermediate points. We will refer to the components which perform this data transfer, between the NICs at the end-points, as *switches*. In some clusters each switch is associated with a particular node; such clusters are referred to as having *direct interconnects*. Clusters that have some or all switches independent of the nodes are referred to as having *indirect interconnects*.

Of particular interest to us are FPGA chips. These have logic that can be reconfigured by the user to suit the application (see (Herbordt et al., 2007b; Herbordt et al., 2008; VanCourt and Herbordt, 2009) for overviews). FPGAs have been applied to great effect for communication, notably in network switches, and also in specialized compute-intensive roles such as signal processing (Liu et al., 2016; Sheng et al., 2015a). Through market forces these applications have brought FPGAs to the point where they now have communication capabilities far exceeding other types of ICs. An FPGA provides up to 100 or so Multi-Gigabit Transceivers (MGTs), each with

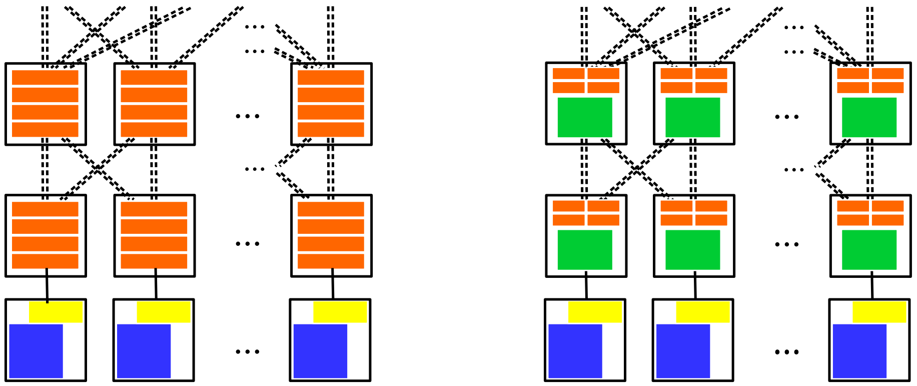


Figure 2·2: HPC system design models I and I'. See text for details.

40 gigabits per second (Gib/sec) of bandwidth and increasing to 100 Gib/sec in the next generation. The majority of the FPGA remains configurable permitting its user (the hardware designer for the HPC node) to freely allocate logic to computation or to communication, in whatever proportion is most suitable.

2.2 Taxonomy of Target HPC Architecture Models

So far we have used an intuitive definition of CiN: computation performed during data transfer among nodes. While CiN capability has been available in some older systems, it is not generally available in current commercial HPC offerings. Exceptions are those consisting of tightly coupled FPGAs (described below). Before continuing the discussion of why other current systems cannot be said to support CiN we first specify CiN more carefully. We do this by classifying HPC architectures and describing how they must be augmented for CiN.

Type I: We refer to a system as Type I if it has indirect communication, i.e., nodes are not associated directly with communication switches (see Figure 2·2) and neither performs functions of the other. The majority of current HPC clusters and supercomputers are of this design, including any installation using Top-of-Rack (TOR) switches to handle all inter-node data traffic.

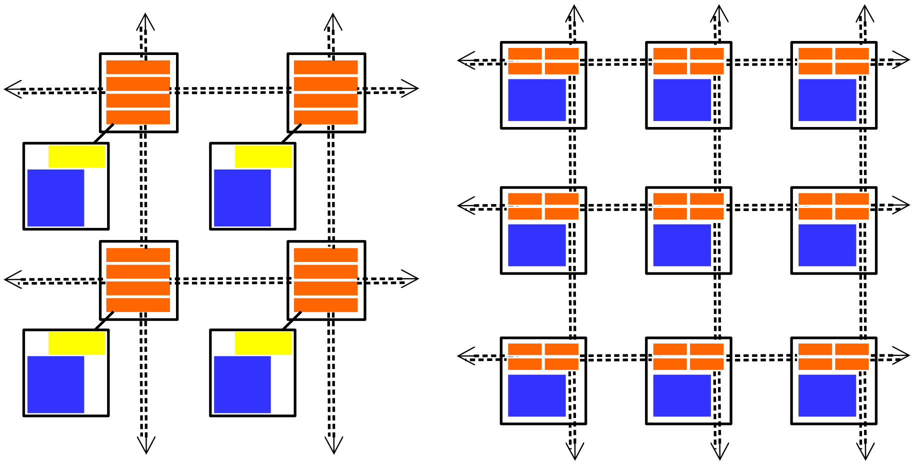


Figure 2·3: HPC system design models D and D'. See text for details.

Type I' is an augmentation of a Type I design to enable CiN through the addition of some computation capability within the switches. The switches have not been changed their basic function. A notable example involves the use of FPGA-equipped Ethernet switches in high-speed trading (Arista Networks, 2013; Agron Design, 2013). We are not aware of any type I' system currently in use for HPC. Here we point out that computing in the NIC does *not* constitute CiN. One example is the Mellanox InfiniBand product line with hardware-assisted collective capability, using circuitry in an ASIC for reduction collectives. A system using such switches is not of type I' because the ASIC is part of the NIC, i.e., part of the node: all network latency is still being incurred in getting the data to the destination node's NIC where the reduction computation is performed.

Type D: A cluster with direct communication is type D. Every node, in addition to performing its own computations, and being a sender and receiver of data, also contains a switch (router) for messages sent by other nodes to other nodes; however the switch operates independently of the nodes' computation logic and does not do anything with other nodes' data except forward it. The history of HPC has included some notably famous examples, such as the Connection Machine (Hillis, 1984);

and some previously-mentioned compute in memory designs like EXECUBE (Kogge, 1994) and IRAM (Patterson et al., 1997).

Type D', named in analogy with type I'. Design type D' is like type D, but some compute capability has been added to the switching logic within each node. That ability is limited, or difficult to access, or the switch and CPU are both on the same IC and the design is using the CPU part of the node to do computation but with a heavy latency penalty from bringing the data out of the switch to the CPU and then injecting it back into the switch for further travel through the network. The Blue Gene/L design, as described at (Gara et al., 2005; Salapura et al., 2005), has all computation on an ASIC which includes network interfaces, and switching is performed by a separate ASIC called the BG/L link chip. The Blue Gene/L uses its 3D torus network for most traffic, as that has the greatest bandwidth. The link chip implements cut-through routing with no need for software intervention, which limits the ability of any interior node to interpose computation upon data transiting through it to and from other nodes.

Type D" systems are like type D, but their nodes incorporate computing logic into the switch in a way that is fully flexible, and is autonomous with respect to the CPU(s). FPGAs provide this possibility in a way that is uniquely flexible, and more efficient than can be achieved with CPUs or ASICs.

2.3 Overview of Research Program for CiN Evaluation

In the taxonomy of section 2.2, the types that are capable of CiN are I', D', and D". In the CAAD laboratory we have type D" systems and subsystem designs readily available for experiments. These have been used to explore CiN methods and measure or estimate their performance.

In any of the direct systems (type D, D', D") the nodes are by definition all of the same design, with each node having a switch, but nodes play different roles with respect to any given inter-node data communication. The node supplying the data and the ultimate destination node are *leaf* nodes, with respect to that communication. Any intermediate nodes that are traversed between source and destination are *interior* nodes. In D-type systems these are full-class nodes on par with the source and destination, but in I-type systems the closest equivalent are switches which constitute a different type of node. In D-type systems nodes will typically be filling both roles (leaf and interior) for most of the time, and shifting roles from one communication operation to another. In a sufficiently large and well-optimized algorithm the nodes will most always serving both roles.

Comparative evaluation of I vs. I' designs, and of D vs. D', presents some difficulties. Typically there is a desire to consider moving applications from one design to another, and a cost-benefit analysis is to be performed. Cost evaluation requires detailed knowledge of the internal design of every part of the system, and in the case of most installed production HPC systems, these details are often not available because they are proprietary or poorly documented. Even with complete knowledge of a design, as is the case for experimental studies in the literature, in order to get the desired benefits, the changes between an I (or D) design and an I' (or D') are so great as to make comparison difficult or irrelevant. An application tuned for one type of system will run sub-optimally or not at all on another type. If a crude analogy of operation exists, there are almost always small differences in load balancing, the mapping of data onto nodes with respect to the system's network topology, and so on. The Department of Energy has addressed this type of issue in their CoDEx (Co-Design for Exascale) (DOE, 2011) program, which develops applications and architectures together.

To perform the cost comparison of an I or D design versus a comparable I' or D' design, we might be able to make adequate approximations from the ratios of the amount of added logic to the unchanged baseline; work on D” designs (with FPGAs and hardware design simulation tools) can inform this.

2.3.1 Focus on D” Systems

We now focus on D” systems and define more of their details. All of the D, D’ and D” systems are direct, but in D” systems the communication (switching) and computation functions are integrated into the same IC; or if separate, the communication functions are in an IC that can be readily reconfigured to add most any type of computation, which is uniquely possible if an FPGA is used. This has been done in Novo-G# (George et al., 2016) and in the Catapult I (Putnam et al., 2014). Both are D” systems with FPGAs directly connected to each other via their MGTs. In these systems the communication logic can be readily changed (e.g., to implement new routing algorithms), and computing logic can be connected directly to the switch/router design.

We now use symbols to refer to the fraction of a node’s logic dedicated to communication and to computing: these are L_{comm} and L_{comp} respectively, with $L_{comm} + L_{comp} = 1$. When necessary for disambiguation, we designate two types of computation, L_{comp_C} and L_{comp_N}, with their sum equal to L_{comp}. The first refers to logic used for any computation performed in the node’s leaf role, i.e., on data that is not in transit to another node. L_{comp_N} is the computation that is essential to our thesis: it is any computation performed in the interior role, i.e., with operands that do not originate on the node doing the computation, and with results that are destined for another node. Note that if in any particular application the FPGA is not actually performing any L_{comp_N} function then the system is operating as a type D system and there is no CiN.

This type of computation, performed in the interior nodes with nonzero L_{comp_N},

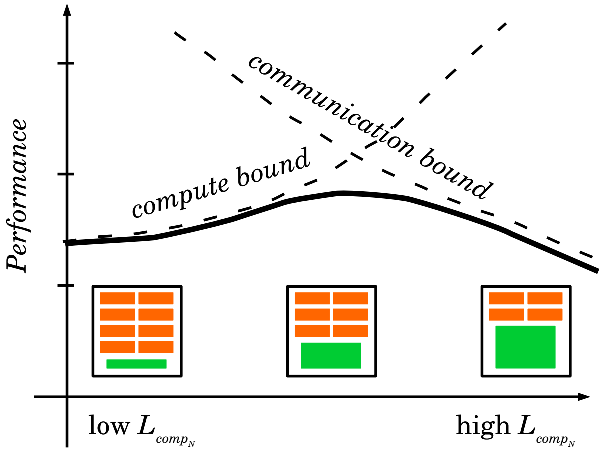


Figure 2·4: Choosing the best ratio of L_{comm} to L_{comp_N} for an application. See text for details.

is the essential service provided by the CiN design model expounded in this thesis. Any computation provided by such logic is what we refer to as *altruistic computation* (AC).

2.3.2 Parameterizing and Exploring Altruistic Computation

The two types of computational logic (L_{comp_C} and L_{comp_N}) are, at the device level, of the same type. In an FPGA that incorporates all three functions, the L_{comp_C}, and L_{comp_N} functions could be designed in a way that shares some component modules or gates. However, this would preclude having a node use the L_{comp_C} function in its role as leaf while simultaneously doing AC through its L_{comp_N} function in its role as interior node. Optimum utilization is our goal, including the ability to perform all three logic functions at the same time in each node.

Given these parameters L_{comm}, L_{comp_C}, and L_{comp_N} and an FPGA-based development platform we can freely propose incremental changes in the three parameters, and in particular, increasing one at the expense of the others. We then evaluate the change in performance; if a shift to greater L_{comp_N} results in greater performance then AC is beneficial. The stylized representation of figure 2·4 shows alternatives for the switching IC of a D" system like that on the right in figure 3·1. The hardware designer can choose how to allocate FPGA logic for a particular application, with varying results. The ideal choice would be near the peak of the solid curve, which is bounded above by the two dashed curves. Different applications would have different dashed curves according to their communications intensity and the extent to which altruistic computation can benefit them; a designer would need to take all of these into account when choosing what L_{comp_N} logic to include.

2.3.3 A Taxonomy of CiN AC Operations

We specifically exclude NIC-type operations from consideration as CiN or as L_{comp_N}. NIC-type operations include packing and unpacking data for transport, encryption, and encoding for low-level protocols. Encryption is a type of computation but is typically employed with paired decryption for transparent end-to-end security, These are functions that are normally performed by the NIC and as such they are understood to be L_{comm}; therefore they are not CiN.

Some operations that we do characterize as CiN include:

1. Operating on data in flight: a detailed taxonomy will be given in the following section.

2. Separate data and control: The sending leaf node does not know what calculation should be performed at the time of sending, but this can be determined at a later time while the data is in an interior node. As an analogy, consider a live database that rapidly updates with new information and responds to frequent (unpredictable) queries. A node receiving a data update must multicast or broadcast, while another node servicing a query must multicast or broadcast with the opcode and return address for the gather/reduction collective that gives the result. This is normally all done in CPU, but CiN offers the opportunity for substantial improvement in certain operations: when the multicasts of the two operations just described

are in the same (interior) node, the opcode can be applied to the update and result sent to the opcode's return address.

3. Load-balancing: the algorithm specifies a distribution of work that turns out to cause some leaf nodes to be compute bound, while interior nodes are largely free. Since all nodes serve both roles, this means that the application is loading the L_{comp_C} logic much more heavily than the L_{comp_N} logic; and we note that this situation often cannot be anticipated as it will be dependent on the dataset or the allocation (provisioning) of physical nodes to a task. If the system has the capability to implement a given operation in either L_{comp_C} or L_{comp_N}, then it could exercise that flexibility in whole or in part, and achieve this third type of CiN.

2.3.4 Sub-taxonomy of CiN Operations On Data-in-Flight

Within type 1 in the previous section, we further distinguish types of operations as follows:

1a. Load-balancing and/or improvement in latency through the use of interior L_{comp_N} in place of leaf L_{comp_C} : if there are N bytes of input to a calculation and an equal amount of output, and the output is only needed at some other node, then L_{comp_C} could be used at either the source or destination; but since it all needs to be sent, one might be able to arrange to use L_{comp_N} to perform some or all of the calculation during transit. Switch buffering or link latency might be such that this calculation's latency could be entirely hidden behind the communication latency. In some cases the computation might even be performed in pieces along the way. This also serves to balance load, as there would be as many as N nodes participating in the computation.

1b. Combining data from two sources to produce one or more results that are forwarded to another node. This includes reduction calculations, such as a collective sum, which is typically done using a spanning tree across all participating nodes with individual calculations at nodes of the tree. As in 1a, this is an optimization because latency can be hidden. This case adds the additional benefit of reducing overall load on L_{comm} by reducing the quantity of data being sent (a collective sum is less data than its inputs).

1c. Systolic algorithms with streams of data traveling in two or more different directions and being used as input and/or transformed at the node where they meet, with data traveling on. The interior nodes are doing calculation, so we are using L_{comp_N}. Data is being sent on to use as inputs to L_{comp_N} on the next node, and/or because they are needed as the destination leaf. As in 1a, the amount of computation might not change from using L_{comp_N} to do it, however there is great opportunity for hiding communication latency. Our case studies in a later chapter fit this category.

1d. Sharing packetization overhead by multiple nodes. Putting data into the network incurs latency associated with the network protocol and L_{comm} implementation. If a leaf needs to send 1 datum each to N distinct destinations, which data also need to be transformed by a calculation as in 1a, rather than sending each datum as a separate message it could be sent as a single block, along a route that hits each destination node, with the L_{comm} hardware instructed to delegate one of the N calculations to the L_{comp_N} logic in each node along the way just before delivery. An example exists in a normalized matrix transpose: a collective sum is used to compute a scaling factor, then all elements must be multiplied by this factor, and all rows (or columns) redistributed among nodes so that the nodes all hold the result in the same row-major (or column-major) order as was the original matrix.

2.3.5 Applicability to D' Systems

The core method of evaluating L_{comp_N} proposals is to consider performance as a function of the ratio L_{comp_N}/L_{comm} or perhaps $L_{comp_N}/(L_{comm} + L_{comp_C})$. This could be envisaged as viewing a graph and choosing the highest point on a curve. In a type

D” system, the entire curve is available at any time (perhaps involving reconfiguration of the FPGAs). The method is also applicable to type D’ systems, despite that only one point on the curve must be chosen when actually building a system.

2.3.6 Ambiguity Between D’ and D” Systems From the Application Perspective

Many applications programming interfaces (APIs) advertise a capability that, as given by its specification, hides details of a system’s capabilities. In the case of distributed or parallel computing APIs, they often make one or more of the types D, D’, D”, and even I, I’ indistinguishable to the programmer. For example, since its earliest versions the MPI specification (Gropp et al., 1998) has allowed the programmer to hand off collective reduction through such functions as `MPI_REDUCE` and `MPI_REDUCE_SCATTER`. The programmer does not need to know or care whether the computation part of the operation is being done by L_{comp_C}, L_{comp_N}, or some of each. This situation resulted from historical (pre-1990’s) work on type D systems with heavily altruistic algorithms; these systems had primitive switching logic and used the CPUs for all L_{comm} functions. To address this issue, our models and our method assume specific knowledge of the system and of the ways that a computation can be carried out via L_{comp_C} and/or L_{comp_N} logic. We can consider multiple alternatives that would be possible on the same system. The actual one chosen for a running application could depend on runtime specifics such as the system’s overall load at the time, number of available physical nodes, and so on.

2.4 Type D” Systems and the Hardware Designer

In describing computations, particularly for collective reductions, there are simple and common operations like those shown in table 2.1. We now point out that in a D” system, all of the L_{comp_N} hardware is reconfigurable and any functions can be defined. We here propose that system designers can be supplied with tools to define these functions from a set of primitives. It is also possible that a L_{comp_N} capability could be customized at runtime, this is addressed in the following section.

2.4.1 Tools for the Hardware Designer

Past work here at the CAAD (Computer Architecture and Automated Design) laboratory includes using an FPGA to implement: bioinformatics algorithms (Conti et al., 2004; VanCourt and Herbordt, 2004; Herbordt et al., 2006; Herbordt et al., 2007a; VanCourt and Herbordt, 2007; Mahram and Herbordt, 2012; Mahram and Herbordt, 2016); machine learning (Geng et al., 2018; Sanaullah et al., 2018); much of a single molecular dynamics simulation application (Gu et al., 2005; Gu et al., 2008; Chiu et al., 2008; Chiu and Herbordt, 2009; Herbordt et al., 2009; Chiu and Herbordt, 2010; Chiu et al., 2011; Khan and Herbordt, 2011; Khan and Herbordt, 2012; Xiong and Herbordt, 2017); complex molecular modeling (VanCourt et al., 2004; VanCourt and Herbordt, 2005b; VanCourt and Herbordt, 2006; Sukhwani and Herbordt, 2008; Sukhwani and Herbordt, 2009a; Sukhwani and Herbordt, 2009b; Sukhwani and Herbordt, 2010; Sukhwani and Herbordt, 2014); Particle-grid mapping (Gu and Herbordt, 2007; Sanaullah et al., 2016a; Sanaullah et al., 2016b); FFTs on the Microsoft Catapult II and Novo-G# systems (the former as a model I cloud computing example, the latter to approach model D") (Humphries et al., 2014; Sheng et al., 2017; Sanaullah and Herbordt, 2018). In most of these, the computing algorithm(s) have required a large and complex hardware design with many parts.

We hope that such designs will be possible in a general purpose HPC system built on model D", with the FPGAs programmable for each client application. In order for an application (such as a molecular dynamics simulation) to benefit from the AC capability of the system, there needs to be a way for the hardware designer to specify the needed calculations and patterns of data movement (including, for example, expansion and reduction trees). In other words, we need an “API” for the hardware designer.

First let us consider an ordinary reduction on a single (scalar) data type. The originating nodes each transmit a piece of data, suitably tagged; routing computation results in the data being routed along many paths that converge at interior nodes, forming a reduction tree. At each node of that tree, there are two or more data merging. The hardware needs to be able to recognize things that are merging as being part of the same computation. For this purpose we assume that the data have already been tagged, specifying which collective computation they are a part of, and what type of merge is being done. The types of computation for reduction would include at least those shown in table 2.1.

This can be generalized to reductions involving non-scalar data (such as vectors) and on structured data (such as ordered tuples with elements of mixed types). Each datum now consists of two or more distinct fields. There now needs to be a way to specify a parse tree describing the structure of the data, and a way to specify which computations are done and in what order. This could be done using code with an expression-like syntax, or possibly though a dataflow programming GUI. A complete GUI-based design system for this sort of application is described in (VanCourt and Herbordt, 2005a).

Many computations that lend themselves well to a systolic array approach can be composed of many individual operations at the grid points, where each operation involves two (or more) inputs coming from different sources (neighboring nodes) and two (or more) outputs with distinct destinations. The outputs are functions of two inputs, that can be defined by parse trees just as with reductions. The only difference is that each node in the systolic dataflow graph has two or more outputs, so there need to be multiple functions. As before, data need to be tagged for disambiguation.

For an FPGA computing system with altruistic capabilities to be maximally useful, there should be tools to enable entire new distributed computing algorithms to be implemented in the L_{comp_N} hardware. Such tools could be used by a person who has a particular algorithm in mind but does not concern themselves with the entire system design.

This can be facilitated by a description language, capable of describing distributed data with complete generality. The distribution of the data across nodes, the type and structure of each piece of data, and methods of determining what gets combined with what, can all be encoded in a language which can be compiled into the needed data structures, defining operations to be performed on data, and defining values for tags to be passed to the hardware by the application or CiN API. As before, this could also be done through a design tool with a graphical user interface, providing the same functions, or a larger development environment incorporating both methods of specification.

2.5 Type D" Systems and the Client Application Programmer

In section 2.4.1 we outlined the process by which new L_{comp_N} capabilities can be added to a system through hardware design. It is also possible to provide a general-purpose L_{comp_N} capability, programmable at runtime, enabling the applications programmer to specify custom or composite computations in a manner similar to OpenGL or OpenCL kernels.

2.5.1 Essential Components of a Design for Computation in the Network

Bringing together the discussion in the foregoing sections, we can itemize the features of a full CiN solution for HPC:

- Nodes each equipped with FPGAs, that can be reconfigured to provide networking with directly linked computation logic (L_{comm} and L_{comp_N}), connected in a direct network to make a D” system.
- An API through which hardware designers can describe computing operations on data in transit (described earlier), for translation to hardware blocks in the FPGA configuration.
- An API through which the client can submit data, describe its type and organization, request one or more operations including communication and computation, and receive the output; but without the need to know where and how any calculation is performed.

In each of the types of AC in the taxonomy of section 2.3.3 the client application submits data to the network and eventually receives results, with the computation being done sometime in between. It is desirable to shield the client application from knowledge of how and where the computation is being done. The client will merely present its data, request the operation, and get the answer.

2.5.2 Useful Scalar Reduction Operations

We can take a hint from the existing MPI standard (Gropp et al., 1998, sec. 4.11.2) and recommend the operations in table 2.1.

Table 2.1: Functions on Typed Data

name	meaning
$(x, y) \rightarrow max(x, y)$	maximum
$(x, y) \rightarrow min(x, y)$	minimum
$(x, y) \rightarrow x + y$	sum
$(x, y) \rightarrow x \times y$	product
$(x, y) \rightarrow x \ \Vert\ y$	logical or
$(x, y) \rightarrow x \vert y$	bitwise or
$(x, y) \rightarrow x\ \&\&\ y$	logical and
$(x, y) \rightarrow x \& y$	bitwise and
$(x, y) \rightarrow\ !!x ==\ !y$	logical xor
$(x, y) \rightarrow x \oplus y$	bitwise xor
$(x_i, y_j) \rightarrow (x > y)\ ?\ (x, i) : (y, j)$	maximum value and location
$(x_i, y_j) \rightarrow (x < y)\ ?\ (x, i) : (y, j)$	maximum value and location

2.5.3 Splitting and Joining, and Multidimensional Data

In an additive reduction, the client could submit a collection of scalars to the network, and get a scalar answer back. These data would ordinarily be submitted from multiple nodes in the role of leaf transmitter, with the answer delivered to one or multiple recipient leaf nodes. However, it should be equivalent for a single node to submit the individual data values as a single vector with n elements, as might be done if the data were already brought together by an `MPI_GATHER`.

The action of joining (concatenating) data, and its inverse (splitting a vector of data into its components) are useful for other purposes. The `MPI_ALLTOALL` function, when all buffers are of the same data type and lengths, with the length equal to the number of ranks, effectively splits each vector of the given data, communicates the pieces, then joins them so that each rank receives one datum from each of the ranks (including itself). Put another way, of the data passed to `MPI_ALLTOALL` are the rows of a square matrix, the results are the rows of its transpose.

It is desirable to provide a way that the client can request operations on multidimensional data, such as matrix transpose and matrix multiplication. To make this possible there needs to be a way that the client can express the way its data (initially split amongst multiple ranks) are to be interpreted as a single object.

A very large matrix may be held in rectangular blocks, one per rank, arranged in such a way that every row and every column of the whole matrix is split up amongst multiple ranks. If each block is of dimensions $a \times b$, and if the whole array is of dimensions $ca \times db$, then there are cd blocks and cd ranks. The partitioning into blocks might need to change for optimal results. If a client task is running as several or many MPI ranks per node, and there is one large FPGA available per node that can perform matrix multiplication, it will usually be necessary to regroup the blocks of matrix data from the block size of the ranks to the block size supported by the FPGAs' L_{comp_N} hardware. As mentioned earlier, it is often useful for the API to hide such implementation details from the client.

2.6 Evaluation Through Case Study

If a hardware system capable of AC via CiN exists, and the necessary design tools are available, the question we would like to ask is, *how much benefit could such a design provide?* Earlier (section 2.3.2) we outlined a general approach involving the relation between the allocation of logic to L_{comm}, L_{comp_C}, and L_{comp_N} functions, and resulting performance of application implementations that optimally use that allocation.

To do this evaluation in practice, we consider case studies of specific computation tasks, that can be implemented on known FPGAs with directly-attached high-speed links and their own routers. In each case we can estimate the performance that might be achieved, and compare to rival solutions using more traditional computing designs.

In the next chapter we proceed to discuss some of these rival computing designs.

Chapter 3

Related Work and Design Considerations

3.1 Other Work Involving FPGAs or ASICs, and Not of Type D”

FPGAs and/or ASICs can be used as networking processors in HPC systems. In most cases these perform only communication-related processing: they transfer application data, but do not transform it nor store it for later re-use.

Exceptions include the latest InfiniBand transceivers from Mellanox (in their Quantum switches) with 200 GiB/sec data transfer rates. The ASIC provides limited computing capability for reductions; this feature is called Co-Design Scalable Hierarchical Aggregation and Reduction Protocol (SHARP). The switch cannot be extended to support new types of reduction or other CiN operations, and its collective functions are available only through drivers via libraries such as MPI (Skjellum, 2017).

The Microsoft Catapult II project (Caulfield et al., 2016) has equipped nodes in Microsoft datacenters with FPGAs for application-specific coprocessing. Each node has one FPGA using two 40 Gib transceivers to directly handle all traffic into and out of the node (the FPGA is a “bump in the wire” between the node’s NIC and the TOR switch). This enables on-the-fly stream processing of the node’s traffic, such as encryption. Unaltered pass-through traffic has very low added latency. The FPGA can also generate and receive its own traffic over either link. Each FPGA can act as an accelerator for its CPU, and many FPGAs in the datacenter can work together as leaf nodes in a distributed computing task. If the task assigned to the CPU is not using its FPGA, then that FPGA is available to other cloud users. As the datacenter has TOR switches and multiple higher levels of switching, it is an I' system. Any computing the FPGAs do will be leaf node computing, so the system is not able to do true CiN operations as a D" system can–for example, in a sum reduction, all arithmetic would take place in the leaf nodes and the communication needs would be the same as if doing the reduction using the CPUs.

Recently Amazon has been offering FPGA-equipped nodes (EC2 F1 instances) on its cloud service. They are very recent Xilinx UltraScale chips with attached memory, but there is no capability for multiple nodes' FPGAs to communicate directly with one another.

Recently Google has been offering their Tensor Processing Units (TPUs) to its cloud customers (Google Cloud, 2017). These are ASICs that are useful for all-to-all and convolutional neural networks for machine learning and deep learning applications.

The Catapult II paper (Caulfield et al., 2016) has a more thorough survey of designs using FPGAs in its *related work* section. None are more applicable to our thesis than those already mentioned.

3.2 An Uncommon Class of HPC Architectures

3.2.1 Proposal

In this thesis we propose to use FPGAs for all of the communication needs of an HPC system, and also utilize additional on-chip capacity to perform certain broadly useful distributed computing tasks, of the types described in section 2.3.3. We propose to do so in a way that avoids the latency and software overhead of prior designs, such as those using existing versions of MPI. Further we propose to invest in significantly

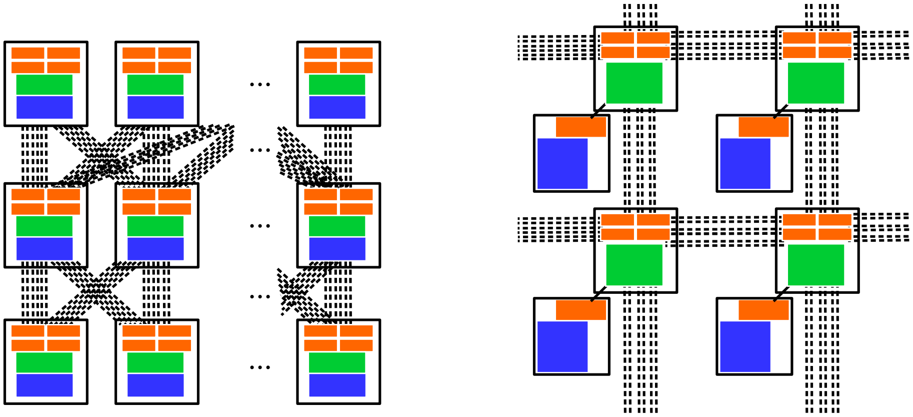


Figure 3·1: Two type D" designs representing the proposal of this thesis. See text for details.

higher bandwidth per link.

The goal is to find higher overall performance through an optimal balance of L_{comm} and L_{comp_N} functions on each FPGA, along with reduced communication latency through tight coupling of these functions, along with increased bandwidth comparable to that of an individual node's main memory.

Figures 2·2, 2·3 show highly schematized representation of three types of HPC systems. Each square represents an IC, assumed to be of comparable cost. Each link is also taken to be of comparable cost. The dotted links represent high speed asynchronous links such as InfiniBand; the solid links are clock-synchronized buses or other high-speed connections0 within the node, such as PCIe. The colors are: blue for standard CPU (computing logic, L_{comp_C}), orange (representing buffers and queues) for communication logic L_{comm}, green for computing logic integrated with the networking hardware for altruistic computation, L_{comp_N}. The physical topologies in figure 2·2 represent conventional multilevel indirect networks such as Clos and N-fly that are common in HPC; the other figures show a grid; but neither topology is specific to models I or D.

Our proposal resembles figure 3·1. Computing capability is added to each node in the IC that handles switching. The two designs shown differ only in physical topology and in the number of ICs used to contain the three types of logic functions L_{comp_C}), L_{comm}, and L_{comp_N}. This is to be done with FPGAs so that the communication and AC logic can be as tightly coupled as possible. There is a much greater bandwidth per link, through the use of many MGTs in each direction. The rest of the node design remains, possibly including a conventional CPU with memory and a GPU or other co-processor. Such designs have been little-studied in recent times.

For the purposes of our case studies we ignore the CPUs and other parts of a node outside the FPGA. We wish to show how much computation could be attained with CiN via a shift of logic from L_{comm} to L_{comp_N}, while the rest of the node is free to perform other work.

3.2.2 Other Work With FPGAs and of Type D"

Prior work of this type includes the Novo-G# system (George et al., 2016), which has been used in earlier work by others here at Boston University. It is a 64-node system with Stratix-V FPGAs performing all the networking functions, as well as computing (in the cited paper, the application is a 3-dimensional FFT). Each link provides 40 Gib/sec of bandwidth in each direction.

The Microsoft Catapult I system (Putnam et al., 2014) had 48 cloud-type nodes each with an FPGA for application-specific coprocessing. The FPGAs have their own network in a 6 × 8 torus topology. Each FPGA has a *shell* with router for communicating with the other FPGAs, a PCIe interface to its host CPU, and DDR3 channels to memory; and a *role* or application area to implement a specific HPC application. This is similar in some ways to what we are proposing.

Our CAAD laboratory is the first to implement wormhole virtual-channel-based routing on a network based on FPGA MGT links (Herbordt, 2018, p. 31).

A recent CAAD Ph.D Thesis (Sheng et al., 2017) deeply explores a D"-type system design similar to that we are proposing. To the extent that our proposal would be used to perform the same task (3D FFT suitable for molecular dynamics), we propose to build on it by increasing the inter-node bandwidth. We are building on that work in other ways, notably by supporting completely different computation tasks. However we share a fundamental design choice with Sheng because, as he wrote, *"the co-location of user logic and router [is crucial for achieving] tight coupling of communication and computation"*.

Sheng discusses a conventional router design that uses virtual channels and performs wormhole routing, such as is described in (Dally and Towles, 2004). Sheng proposes the addition of pipeline stages to accommodate collective operations (such as multicast and reduction), which we would include in order to maintain versatility. He also proposes an enhancement specifically for Novo-G#, involving *injection ports* and *ejection paths*. These are for leaf traffic, and for messages that traverse only one link they bypass most of the routing logic, reducing latency.

3.2.3 Improvements to Bandwidth and Latency

The latest FPGAs support on the order of 96 MGT channels per chip. If connected in a 3D grid, each FPGA could have up to 16 channels per grid direction. Each MGT can operate as about 20 Gbits/sec, giving about 2 GiB/sec per channel or 32 GiB/sec overall per direction. This is of comparable magnitude to the bandwidth of DDR channels to memory. For example, the Intel Xeon E7-4830V3, a 12-core Haswell-EX product (Intel Corporation, 2015b), has a total memory bandwidth of 85 GB/sec.

Turning to latency, the MGT channels incur a latency of about 100 ns. As seen in figure 4·5 traffic originating on-chip bypasses some of the routing logic and queues, as does traffic destined on-chip. So, routing adds only a few cycles when the communication traverses only a single link. Therefore, the expected latency will be comparable to that of a DDR memory device.

The grid or torus physical topology is suitable to some classes of problems, but not all. We accept this limitation but point out that our proposal is equally applicable to any physical topology.

3.2.4 Differences From Prior Work

This proposal involves a single chip type performing an entire task, such as Matrix-Matrix Multiply (MMM). Systems with this attribute were once popular (most notably in the late 1980s and early 1990s), when they were called single-chip glueless scalable designs. There are a few important differences here.

Out thesis does not propose to do any HPC task completely in the FPGAs; rather the FPGAs are primarily there to perform the communication processing, but are augmented with the capability to perform certain computational tasks as well.

Our thesis does not deprecate multi-IC node designs or mandate a transition to single-IC designs; to the contrary, the standard CPU is assumed part of the design as it is general-purpose with memory and possibly such things as PCI-attached storage or a GPU. The Microsoft Catapult I (Putnam et al., 2014) system approaches our proposal, but we propose much greater bandwidth for inter-FPGA communication.

Older glueless designs used synchronized clocks, and usually a system-wide synchronous broadcast of some kind (as in early massive-scale SIMD systems, such as (Hillis, 1984), in later special designs e.g. (Kogge, 1994)). Present-day GPUs are SIMD-like, with many stream processors within a single chip. Our proposal does not rely on or propose global synchronization. However we do place a heavy emphasis on avoiding variations in latency to improve overall performance.

3.2.5 General Distributed-Computation Design Considerations

Routing Without Bubbles : In an older “telephone model” for computer networks, a point-to-point connection is established and bandwidth is guaranteed: once the first unit of data arrives, the rest of the data stream can be sent without bubbles or gaps between data elements. This would be possible if a number of conditions hold: 1) The partitioning of the application maps onto the network graph in such a way that all communications traverse only a single edge; 2) this mapping does not change (no node failures or task migration); and 3) any given data stream is completed before another one starts. For the purposes of the following analysis, the first is the most important because it avoids two simultaneous data streams competing for any link.

Graph Emulation of a 2D Grid : A system might not actually provide a 2-D grid, because of its network topology, or because of the unavailability of a subset of nodes that are connected in a grid. In this case the application designed to use a 2-D grid pattern of communication will still use such a pattern, but actual network traffic will follow some other pattern. The physical network is said to be *emulating* the application’s desired topology. Messages will need to traverse two or more edges of the network graph, greatly increasing latency as compared to the ideal mapping in which the physical network matches the application’s needs.

To mitigate this problem, HPC systems often allocate nodes to tasks in a way that tries to map the application’s desired graph onto the physical graph in a way that is optimal with respect to latency.

Our MMM and sparse matrix inversion examples use algorithms that can be done efficiently on a 2D grid. We assume that a 2-D grid can be emulated on the actual network, with never more than some small constant (perhaps 2) physical hops per emulated edge. We also assume that full bandwidth will be available over each emulated edge.

There are many available algorithms for MMM on gridlike-connected many-node systems, see for example (Li et al., 1993), (Geijn and Watts., 1997), (Gunnels et al., 1996). We choose to use contiguous blocks of matrix data, and the Cannon algorithm in the variant that moves A and B data while keeping C stationary. This avoids group broadcasts (such as row-wise broadcasts of blocks in a single column); and within the broadcast-avoiding methods it requires the fewest number of transfers overall.

Memory Capacity : FPGAs often provide one or two hard IP (non-reconfigurable, fixed, speed and power-optimal design) DDR interfaces, and newer ones provide the ability to control several or many channels. Stratix 10 products support several DDR memory interfaces (limits depend on the specific product, design limitations of power usage and speed (Intel Corporation, 2018), and overall system cost). Having many channels and many attached memory devices will contribute greatly to the cost of the node. We consider this to be a sufficiently great disadvantage that it is not worth pursuing further. A single DDR interface may make sense; more than that would overly narrow the applicability of our study.

Given a limit to at most one DDR interface, it cannot contribute much to data access bandwidth. By contrast, there are many small memory blocks on the FPGA itself, each of which can be addressed independently and each providing high bandwidth. The designs considered here do most (or all) of their data access via on-chip memory.

Memory Bandwidth and Communication via MGTs :

A DDR4 channel at 2333 MHz provides 18.7 GB/sec of throughput and various latencies (depending on access patterns and controller design, see (Song and Parihar, 2012) and (Chang., 2017); for our purposes 10ns-100ns is a good guideline). These are comparable to the use of 8 MGT transceivers in parallel. With 16 MGTs per edge of a 2D or 3D grid/torus topology, nodes can receive data from a neighbor at a higher sustained rate than they could read that data from locally attached DDR4 SDRAM.

FPGA Utilization and Efficiency Trade-offs :

FPGA design has many inter-related restrictions. They often occur in the form of trade-offs: for example, a designer might find that not all of the device's DSP units are available because there are not enough general-purpose connection buses to place and route the specified design. They would then need to either reduce the scale of the design (and not use all of the chip's DSPs) or change to a design that uses a less demanding connection scheme. Thus, there is a trade-off between DSP unit utilization and connection flexibility.

Many place-and-route and timing limitations are too complex to work out directly; instead one must simply try a design, see what does not work, then change the design and try again. This situation is exacerbated by a very slow development cycle (typically many hours to place and route a design). Some of these difficulties are cited in (Baxter et al., 2008), which describes experiments on a grid FPGA supercomputer with a design similar to that we propose. They report place-and-route times on the order of six hours. The situation hasn't improved in ten years; here in the CAAD lab we are seeing times of 18 hours or more.

In our estimates we assume that the FPGA and design tools will enable routing all signals in any reasonable design that employs fewer than 90% of the DSP units, and that long signal paths will need to be registered, adding a few or several clock cycles of latency. Here we make a similar assumption with regards to ALMs (general-purpose logic) and memory blocks (BRAMs and lookup tables).

Power Efficiency : As outlined in section 3.3.2, the thermal budget of the latest FPGAs is comparable to the latest high-performance CPUs. Depending on DRAM utilization and other factors, this proposal could double the power usage of each HPC

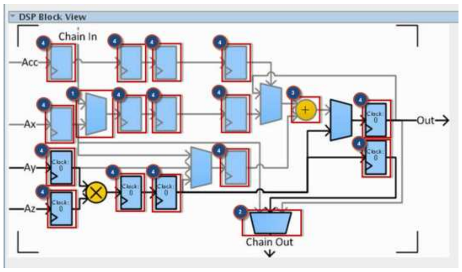


Figure 3·2: A Stratix 10 DSP block (from (Intel Corporation, 2017b))

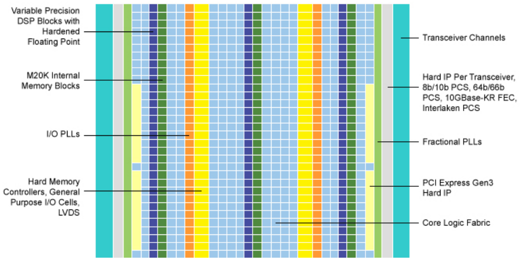

Variable Precision DSP Blocks with Hardened Floating Point
M20K Internal Memory Blocks
I/O PLLs
Hard Memory Controllers, General Purpose I/O Cells, LVDS
Transceiver Channels
Hard IP Per Transceiver, 8b/10b PCS, 64b/66b PCS, 10GBase-KR FEC, Interlaken PCS
Fractional PLLs
PCI Express Gen3 Hard IP
Core Logic Fabric

Figure 3·3: One possible arrangement of DSP blocks on an FPGA (from (Linux-Gizmos, 2017))

node when working on a particular task. This would be partially offset by the use of the FPGA, and not an ordinary NIC, to provide all networking functions. It is also important to ensure that functional units within the FPGA design can be shut down when not in use to save power.

3.3 Product-Specific Design Considerations

In all cases we treat the Stratix 10 family as an example of FPGA designs in general; many other general-purpose FPGAs will have all of the constraints described here, differing only in details.

3.3.1 Stratix 10–DSP Blocks

Figure 3·2 shows a typical FPGA hard compute unit: the DSP block in the Stratix 10 series. Note the ability to accumulate a new product into a sum in a single cycle: streaming two vectors into the A_y and A_z inputs will result in their dot-product on the output 5 cycles after the final vector elements are input.

The design shown process data in IEEE 32-bit floating-point format. This is one of several user-accessible configurations that are all implemented by the same underlying hard IP. Another option provides two multiply units, but in fixed-point representation and at lower precision.

3.3.2 Stratix 10–Communication Links

Power Requirements of MGTs : It is unclear whether all 96 MGTs in a Stratix 10 could be used at the same time, while also using a significant fraction of the DSP and memory blocks. The answer will depend on clock speed as well as the cooling system. We are using 300 MHz as a design clock frequency. Intel/Altera provide some power estimation guidelines (Intel Corporation, 2016) that suggest that the memory blocks will use about 7 W, the DSP blocks 8.6 W, all 80 MGTs operating at 17.4 Gbps would use 45 W; all with the core clock at 500 MHz (which itself should use about 6W). The estimated total TDP would be around 70W; the package size and design are similar to recent dual-socket Xeon CPUs, which have similar or higher TDPs (Intel Corporation, 2017c).

Impedance Matching : To operate at full speed, the MGTs need to be configured to match the impedance and other electrical characteristics of the interconnect cables. This configuration can take quite a bit of effort with individual adjustments per link; however it helps to match cable types and lengths whenever possible.

Bandwidth : The bandwidth of a single MGT channel can exceed that of a BRAM's write port. For example, in much of the work of (Sheng et al., 2017) the core clock was 150 MHz and MGT clock was 75 MHz, with one phit (physical digit) of 256 bits being transmitted each MGT clock. The BRAM word width is 64 bits, so if a single BRAM port is used (the other port is needed for other purposes) the MGT bandwidth is 4 words per MGT clock and 2 words per core clock. If the BRAM is clocked at the core clock speed, it would require the use of two BRAM blocks or *slices* to keep up with a single MGT's receive data stream.

Latency : In (Sheng et al., 2017) is a statistical study of latency of the Stratix V MGTs, using a phit rate (MGT clock) of 75 MHz. The mean latency was about 13 clocks, and standard deviation about 1. The core clock (used for the router logic) was twice the speed of the MGT clock, and routing required 7 clock cycles. Allowing for a $4.5 - \sigma$ departure from the mean, latency can be estimated to be $13 + 4.5 + 7/2 = 21$ MGT clocks, or about 280 ns.

Jitter : Each multi-gigabit link uses two clocks, one in each direction, each synchronized with the sending end's clock. These clocks cannot be perfectly in sync, so the rate of data flow in each direction through a given MGT will vary. This issue is handled inside the MGTs on the sending end by deliberately inserting a filler phit at intervals, and on the receiving end by discarding these.

For example, if a node is sending 2048 phits through its MGT and the core clock rate is exactly twice the MGT clock rate, it will take 4096 core clock cycles to send all the data; but in the same time the number of phits received from the other end might be 2047 or 2049. Also, if two blocks of data is sent out over different ports, and two others being received, the transfers will take different amounts of time even if all four blocks are the same size and there are no routing/buffering delays in the network.

Therefore, it is important that the controllers responsible for generating outgoing data streams and handing incoming data be independent from each other and from the controller(s) doing local calculation. Double-buffering will be used (to calculate on one block of data while the next is being received), so the master controller may need to wait for all transfers to complete and for local computation to complete before attempting to switch buffers or initiate new transfers.

3.4 Considerations Specific to Most Problems

3.4.1 Granularity of On-Chip Memory

Stratix-10 (and any FPGA family/architecture) has a limited number of memory partitions. In the higher-end variants of Stratix 10 there are over 11,000 “M20K” blocks (each has 20,000 bits), which can be combined to make larger blocks. This is not enough to have, for example, two memory blocks for each DSP block (unless no memory were being used for anything else!). It is unlikely that desired memory sizes will be an exact multiple of the M20K blocks’ size. Designs must take these into account by allowing for a large fraction of BRAM capacity to remain unused.

Chapter 4

Case Studies

4.1 Method

It is beyond the scope of this thesis to create a whole FPGA design including a router design like (Herbordt et al., 1999; Sheng et al., 2014; Sheng et al., 2015b; Sheng et al., 2016b; Sheng et al., 2016a; Sheng et al., 2017; Sheng et al., 2018) along with one of the applications described below, and get a many-node configuration running in simulation.

To estimate the performance of a multi-FPGA algorithm, we work out how the task can be split across multiple nodes, then estimate the time taken for the local computation on an individual FPGA, and estimate the time for each FPGA to send data to neighboring nodes. These estimates are then combined in the appropriate way (with overlap in time, when possible). All of the considerations described above (such as latencies of the DSP blocks) are taken into account.

4.1.1 Dense Matrix-Matrix Multiply

As we mentioned in section 3.2.4, we chose a distributed dense matrix-matrix multiplication technique that avoids row-wise or column-wise broadcasts and communicates only between adjacent neighbors on a 2-D grid. The local computation consists of dense matrix-matrix multiplication of two blocks of A and B (treated as smaller matrices) to yield a block (matrix); several of these are added together to form that node's share of the final answer C. With extra buffers, each local MMM can be done while the next blocks of A and B are being sent.

4.1.2 Single-FPGA Dense Matrix-Matrix Multiply

In the matrix multiplication $C = A \times B$, the elements of C are dot-products of rows of A with columns of B. The computation of a dot-product of two k-element vectors can be viewed as a dependency tree, with a minimum latency of $\lceil \log_2(k-1) \rceil$ times the latency for addition plus 1 times the latency for multiplication. However a full add tree would consume at least k DSP units for each k-element dot-product that we wish to compute in parallel, and most of these resources would be idle for most of the time, unless (for a massively-parallel SIMD or systolic implementation) the memory for all three arrays A, B and C were partitioned into at least as many slices as we have add-multiply trees.

Instead, a single DSP unit can compute the k-element dot product with a latency of $k-1$ additions and 1 multiplication if it performs each multiplication serially. If the product C is of dimensions $n \times m$, then all nm dot products could be computed in parallel using nm DSP units, if the A and B matrices are stored in BRAMs that are partitioned (sliced) by rows and by columns respectively. The nm DSP units would each have its own minimal-size BRAM to store a share of C.

In ideal form (which is not possible, as described below) is illustrated on the left side of figure 4·1. On the jth clock cycle, the i BRAMs holding the i rows of A would each broadcast one element A_{ij} to all of the DSP blocks in the corresponding row of C; simultaneously each of the BRAMs holding the columns of B would broadcast the jth element to the corresponding column of C. The entire process would take k cycles of broadcasting, plus the latency of the first multiply-add, to get a dot product C_{ij}, a single element of the answer.

For computing products of large arrays we need to store more than a single row of A and B per BRAM block, and have DSP blocks computing more than a single

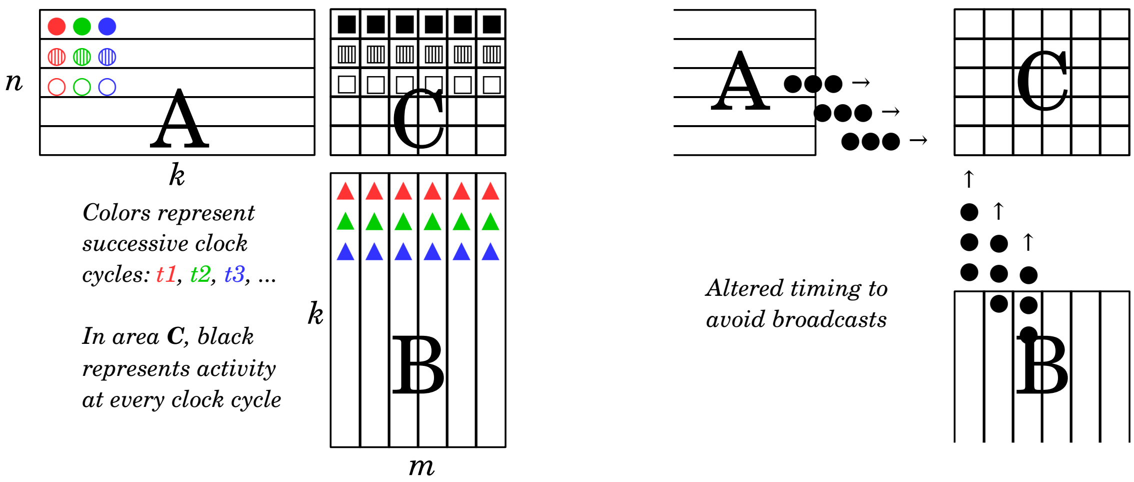

n
A
k
C
Colors represent successive clock cycles: t1, t2, t3, ...
In area C, black represents activity at every clock cycle
k
B
m
A
C
Altered timing to avoid broadcasts
B

Figure 4·1: Single-chip matrix-matrix multiply using a large array of DSPs performing multiply-accumulate. a) depicts *idealized* operation using broadcasts of A elements to an entire row, and of B elements to an entire column, of the DSPs. b) alters the timing so that elements of A and B travel one cell at a time through the systolic array.

element of the product C. With a suitable controller we can hide most of the latency of the multiply-add, and of the propagation through the systolic array, which we address next.

4.1.3 Data Broadcast Timing

In practice, large broadcasts are a bad idea in FPGA design. The DSP blocks on an FPGA are in a fixed layout that is meant to be fairly optimal for the majority of user designs, and that means it will not be conveniently arranged in an $n \times m$ grid for us.

It is fair to guess that if a design attempts to broadcast A_{ij} to all of the DSP blocks in the corresponding row of C, the longest path in the broadcast distribution tree will be a major fraction of the distance across the chip, traversing many junctions of the interconnect fabric, each adding gate-delays. Therefore we assume that a single-cycle, or even a “few”-cycles row-broadcast is impossible. (The handling of a core clock is a separate case: FPGAs have special clock distribution trees that bring a synchronized clock to all parts of the chip, most user designs require one or more global clocks. See e.g. (Altera, 2017))

Altera and Intel (Intel Corporation, 2015a) have made a big deal about the HyperFlex architecture and Hyper-Registers that are used by the design tools to automatically pipeline long chains of combinatorial logic and help with long paths through the FPGA's interconnect. Though a single-cycle broadcast is likely impossible, a "few"-cycles broadcast latency-matched to all recipients would probably work, and that could just be added to the existing five-cycle multiply-add latency of the DSPs.

4.1.4 Systolic Array MMM Designs

However, we can instead use a classical systolic approach in which no data element moves more than one cell per clock cycle. This makes the overall design easier to place and route, makes this analysis more definitive, and does not incur significant loss of performance, because after the first k cycles the extra latency can be hidden by starting the next set of ij dot-products while the previous one is completing. Each cell of the systolic array is one BRAM and one DSP block and is working on a single element of C at any given time; elements of A travel horizontally through the array while elements of B travel vertically.

The overall multiplier design is shown in figure 4·2. Each clock cycle, one element of A is read from each of the slices of A BRAM and fed to the leftmost element of a row of the systolic array. At the same time, an element of B is read from each B slice and fed to the bottommost element of each column.

The rows and columns each need a controller. This controller decides which BRAM element to read out on the current clock cycle, and when to tell the systolic cells to write their result (sum), reset the running total to zero, and/or start on a new element of *C*. If one of these special things needs to happen, the message needs to be passed along to all the cells in the systolic array. This is done with control signals generated

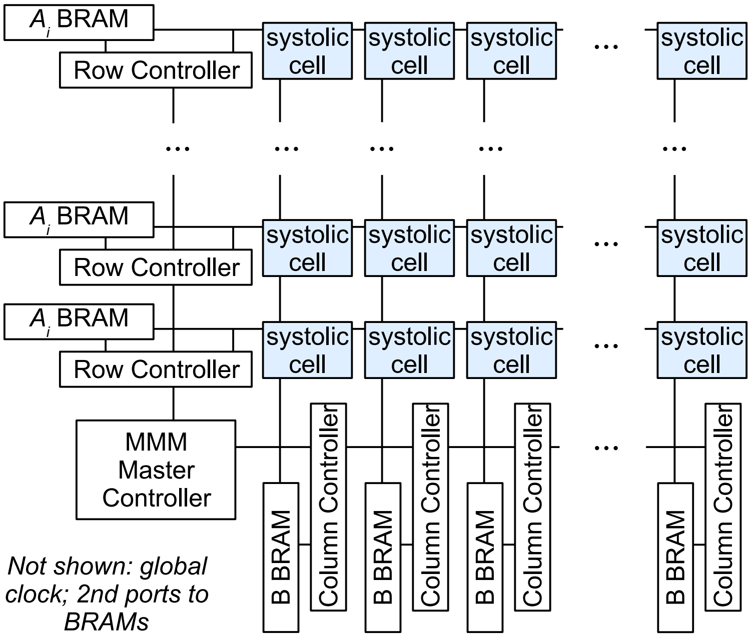


Figure 4·2: Systolic array Matrix-Matrix Multiply. Details in text.

by the row controller.

Each cell of the main systolic array is as shown in figure 4·3. On each clock cycle, a value from array A and a value from B arrive from the left and from below, respectively. In addition, from the left come control signals telling when to start working on the next element of C. Typically, the A and B values will get sent to the inputs of the DSP block, which is operating in IEEE 32-bit multiply-accumulate mode. The DSP block outputs its current sum, and this sum lags behind the input by five clock cycles.

Each slice of A has a BRAM slice and a row controller, shown in figure 4·4. The row controller receives an opcode, number of columns, and current row index from below; on each clock cycle it passes these values to the row controller above it. When the row controller is told to begin a row, it resets its column counter to zero. On each clock cycle it reads a data value from BRAM, which gets sent into the row of the systolic array. When the column counter reaches the maximum, the controller stops

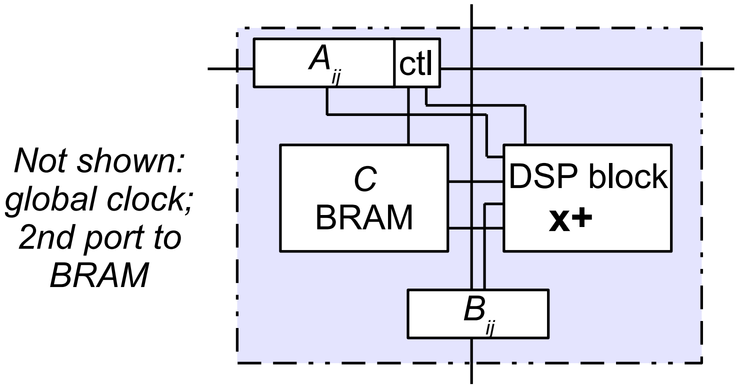


Figure 4·3: One cell of the MMM systolic array. Details in text.

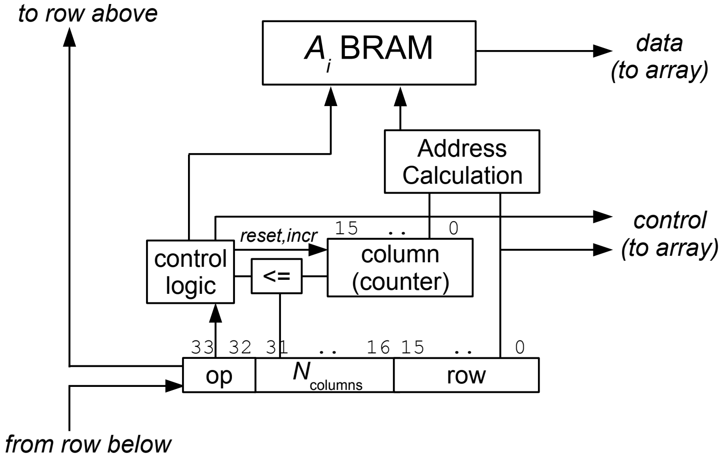


Figure 4·4: One row controller. Details in text.

fetching new data from the BRAM but waits five more clock cycles before sending the control signal into the systolic array telling it to write sums back into the C BRAMs.

4.1.5 Flexibility of Design; Choosing Dimensions for Simulation

The DSP blocks in a modern FPGA are not clustered together in a single block; long columns such as in figure 3·3 are more typical. In order to maximize efficiency in place-and-route (both in the quantity of logic modules, DSP units, etc. needed for the design, and to achieve the highest possible clock speed) it is likely that a rectangular array would be preferable to a square systolic array. There is no special reason a square is needed, though aspect ratios closer to 1.0 are better because they reduce the $\max(n, m) - 1$ latency discussed below. There is no change in communication bandwidth needed for the multi-FPGA design, so long as the entire problem is broken up into an integral number of rows and columns of equal size. This can be achieved by adding extra columns/rows filled with zeros if needed.

For computing products of large arrays we need to store more than a single row of A and B per BRAM block, and have DSP blocks computing more than a single element of the product C. If the BRAMs for A and B are large enough, the number of rows per slice of A can be the same as the number of columns per slice of B, and in any case this will result in the same-sized BRAM blocks because the other dimension is still k. So, without loss of generality we can use a rectangular systolic array of any aspect ratio to efficiently perform the $C = A \times B$ matrix multiplication for any values of (n, k, m).

Therefore, for the rest of this discussion we will take $(n, m) = (32, 32)$ and $i = j = k = 1024$, we are multiplying two 1024×1024 square matrices. There is a grid of (32×32) cells, each consisting of a DSP block and its associated BRAM, holding a $1/(32 \times 32) = 1/1024$ share of the product array C.

4.1.6 Time for Local Calculation

Each row of the grid is fed by BRAM that holds a 1/32 share of the rows of array A; in the example these would each hold 32×1024 elements. Elements of A travel horizontally through the row, one cell per clock cycle. Similarly, each column is fed by BRAM holding a 1/32 share of the columns of array B, 1024×32 elements in all, and data travel vertically one cell per clock.

The design described in the previous section delays rows of A and columns of B in such a way that matching pieces of data reach each cell of the systolic array at the same time; see the simplified sketch in figure 4·1. In the figure, the bottom row of A

and the left row of B can begin on the first clock cycle; the next row and column can begin one clock cycle later, and so on. The maximum delay will be one less than the number of rows or the number of columns in the systolic array, whichever is larger. Thus the latency to the farthest corner (upper-right cell) will be $\max(n, m) - 1$, i.e. 31 clocks in this (32×32)-cell example.

Within the DSP block there is a 4-cycle latency to get the first pair of data through the multiply unit and adder, and one more cycle for each additional pair of terms whose product is added to the running total.

Each element of the final product matrix C is computed in $L + 1024$ consecutive cycles, where L is the total latency $(31 + 4$ in this example), but they aren't all computed during the same $L + 1024$ cycles. With a 32×32 systolic array we can compute 32×32 elements of C in $L + 1024$ cycles. The node's share of C is likely bigger, and if so, the DSPs will each begin on another element of C and the appropriate rows of A and columns of B will be streamed out of the BRAMs. For these subsequent phases of the operation, most of the $31 + 4$ cycle latency can be hidden by starting on the next A row and B column shortly after the previous ones have finished. There still need to be a few cycles for the DSPs to get the full total for writing back to the C BRAM, then to clear their running total to zero to start a new dot product.

Because each cell in the systolic array holds a $1/(32 \times 32) = 1/1024$ share of the product array C, the entire calculation of $C = A \times B$ would take 1024 of these $(1024 + L)$-cycle phases. Using a highly conservative 150 MHz core clock, this design would take 7.0 milliseconds to compute the product of two 1024×1024 matrices.

Intel/Altera claim 9.2 TFLOPs (IEEE single precision) for the Stratix 10 2800 products, which have 5760 DSP units; this means they're counting on all being utilized and a clock speed of 800 MHz.

We have been assuming that only a fraction of the DSP units will be available and a much more modest clock speed. Others in our research group who are familiar with the Stratix 10 have advised us that it's reasonable to estimate a core clock of 300 MHz would be attainable, but not much beyond that. At 300 MHz, the time for the 1024×1024 MMM drops to to 3.8 msec.

It is also reasonable to expect that a larger than 32×32 systolic array should be achievable; up to 5760 DSP blocks are available. Using a 64×64 systolic array brings the time down to 0.87 msec, and floating-point performance to 2.46 TFLOPs.

4.1.7 Dense MMM on a Grid of FPGAs

As we mentioned in section 3.2.4 we chose the Cannon algorithm with stationary matrix C to organize the movement of blocks of matrix data between the nodes on a grid. This algorithm is described in (Li et al., 1993). To summarize, the elements of A and B are initially distributed evenly among the nodes of a 2D grid; there is an initial setup phase in which row i and column j need to be rotated until all pieces of A and of B are on the same node as a corresponding piece of C that depends on both of them; and thereafter the calculation is done concurrently with simultaneous single-step row- and column-rotations until every dot-product in C is complete, which can be done without any computing elements being idle.

We assume that the FPGA design includes a wormhole, virtual-channel based router similar to that shown in figure 4.4 of (Sheng, 2017). A block diagram is in figure 4·5, showing just four pipeline stages (routing computation, virtual channel allocation, switch allocation, switch traversal). That work develops the router design further to include three more stages to handle collective operations, but we have left these out of the diagram for simplicity.

Above in section 4.1.6 we discussed BRAMs holding “slices” of arrays A and B; in the example there were 32 slices for each.

For the multi-FPGA design these BRAMs would need to be doubled in number,

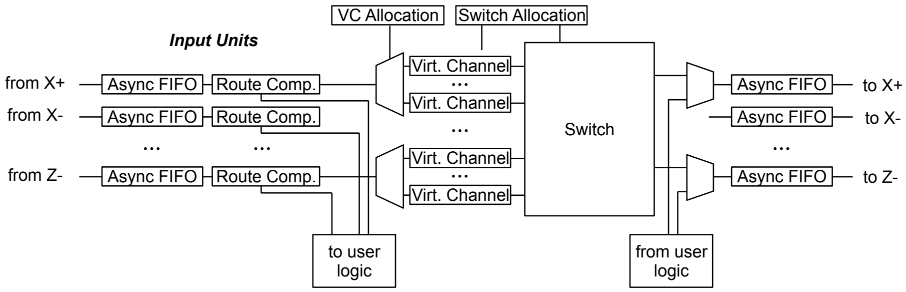


Figure 4·5: Router design suitable for our proposal. Adapted from (Sheng, 2017).

but each slice remaining the same size as before, to hold two equal-sized blocks of A or B data. One of these BRAMs is in write mode, storing data being received from a neighbor node, while the other BRAM is in read mode on both of its ports, transmitting its contents to a neighbor node while simultaneously supplying data (usually with a different access pattern) to the calculation units.

Data must travel between the router and the BRAM slices of A, B, and C. For the same reasons that we are using a systolic array for computation, we must use a design for reading and writing the BRAMs that will avoid long paths through the interconnect. This can be done by a mechanism like that in figure 4·6. There will be circuitry like this for all five sets of BRAM slices (two of A, two of B, and C).

Suppose one set of BRAM slices is currently the receive buffer for a block of A data from the router. A controller will monitor the receive port(s) of the router, waiting for the appropriate packet to arrive. Each time payload data arrives, it will place that data, along with an appropriate address and control bits indicating *write*, to the register **address data R/W** in the figure. An address decoder looks at the address to see if it applies to any of four BRAM slices under its responsibility. If there is a match, it asserts the select line of the BRAM and gives that BRAM the needed address bits and R/W control signal. The BRAM reading data asserts its data output, and the multiplexer (also controlled by the address decoder) routes the data to the **address data R/W** register on the bottom. That register leads to similar logic for another set of BRAM slices.

Operation is similar in the case of reading data and streaming out to the router. In this case, there is also the possibility of back-pressure from the network's flow control. If at some point the router cannot accept more data, the process of reading data from the BRAMs will have to pause. For this reason, the controller responsible for transmit-receive needs to have a buffer large enough to hold however many data reads might be in flight. In the example shown in the figure the BRAMs are in groups of 4, and for this discussion we have been considering a total of 32 BRAM slices per set; so there might be as many as eight **address data R/W** transactions in the pipeline. There needs to be a buffer that can hold this much data. The same buffer will also be used for assembling data into the flits (flow control digits) used by the router, for example combining four 32-bit floats into a 128-bit flit.

We now point out that in the single case of the C BRAMs, there are more and smaller slices. A similar design will work, but there is potentially more latency. It would also be feasible to distribute the C BRAMs among several groups each with its own **address data R/W** transaction pipeline, bringing the latency back down to a level comparable to the others but adding some complexity to the master controller for C BRAMs.

There is the possibility, mentioned in section 3.3.2 under the heading *Bandwidth*, that it might take more than one BRAM block to keep up with the MGT rate. This might mean slicing the BRAMs up to a greater degree than discussed already. In this case sets of BRAMs (each with its own read-write transaction pipeline and controller) can be associated with MGTs via multiplexers and/or demultiplexers. In a 32×32 systolic array example, if we use 4 MGTs per grid direction then each would be matched to $32/4 = 8$ pairs of BRAMs.

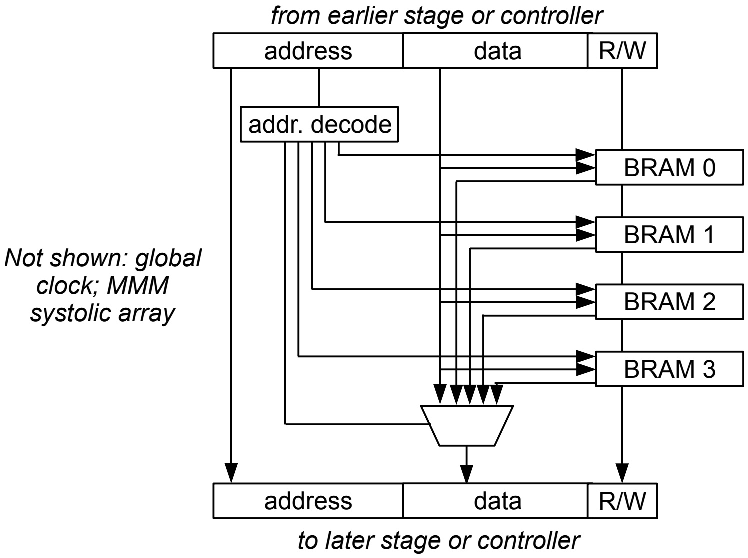

from earlier stage or controller
address
data
R/W
addr. decode
BRAM 0
BRAM 1
BRAM 2
BRAM 3
Not shown: global clock; MMM systolic array
address
data
R/W
to later stage or controller

Figure 4·6: Writing a data stream to, or reading a stream from, BRAMs organized as slices. Details in text.

As discussed earlier the latency in MGT clocks for sending data through one link in the network is about 20 MGT clock cycles. Different router and buffer designs are described (Sheng, 2017); we are including latency for: the MGTs themselves, small FIFOs to deal with jitter, and a 7-stage router which includes stages to handle collectives. These are MGT clock cycles or phit cycles. Throughput is one phit per cycle, the phits are 256 bits = 32 bytes = 8 data elements; the MGT clock speed is 75 MHz.

4.1.8 Simulation for Performance Estimation

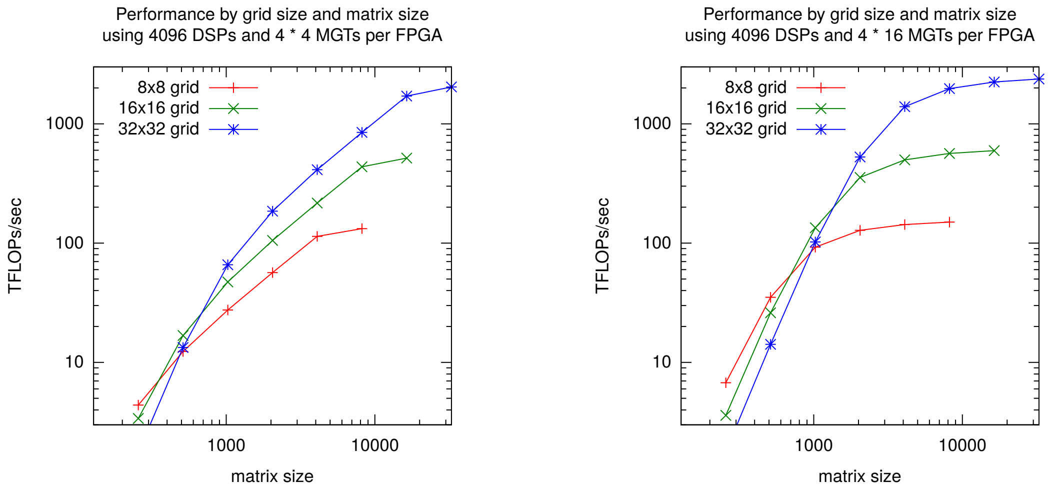
We created a program to simulate the operation of a multi-node distributed matrix-matrix multiply, and another program to calculate the time it would take, incorporating all of the details as discussed above. This second program works out the times needed for communication and for local computation, then computes total time based on the constraint that both need to finish before the next stage of the Cannon algorithm can proceed.

To illustrate this by example: If the local blocks of data are 1024 × 1024 in size, there are $1024 \times 1024 \times 4$ bytes $= 128 \times 1024 = 2^{17}$ phits of A data to transmit to the west while getting the same amount from the east. Using a single MGT, the communication would take 2^{17} clocks at 75 MHz, which is 1.7 milliseconds. The system would be compute-bound if each FPGA has a 32×32 systolic array running at the clock speeds considered earlier (7.0 or 3.8 msec of compute time). With a 64×64 array, the local calculation would proceed 4 times as quickly (1.75 or 0.95 msec) and so the system would be roughly evenly balanced if using the slower clock speed, or communication-bound with the faster clock speed. We will elaborate more generally on when the system is compute- or communication-bound in the next section.

4.1.9 Estimated Performance

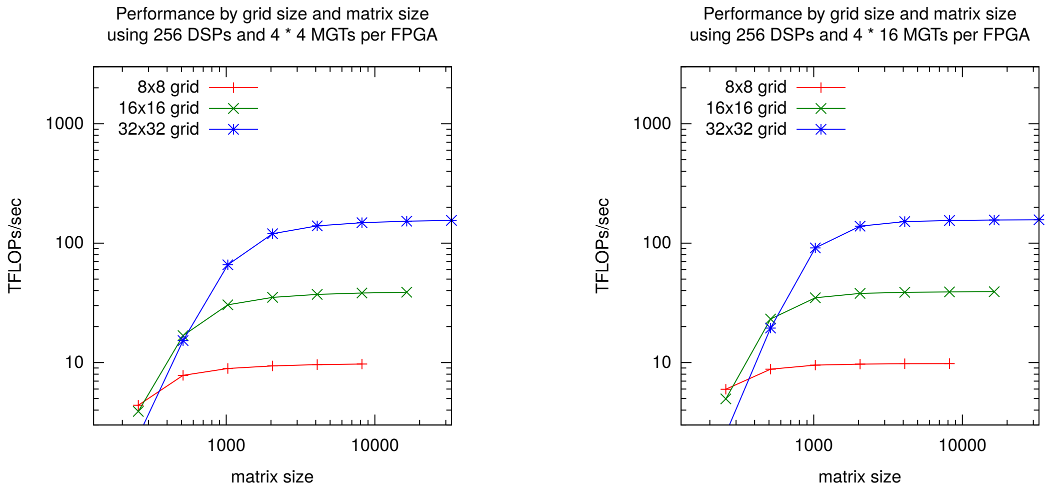
In figures 4·7 through 4·7 are shown four corners of the HPC FPGA design space: varying the on-chip computing capacity on the one hand, and communication capacity on the other. Within each chart there are plots showing performance versus problem size for three choices of the system's grid size (i.e. the number of FPGA nodes in a 2-D torus network).

In figure 4·7 (a) there are 4096 DSP units and only 4 MGTs per grid direction in the 2D grid. In all cases shown, the computational power of the systolic array is so great that the whole system is communication-bound: It takes longer to transmit the data from one node to the next than it takes for each node to do its local computation. As the problem size increases (moving to the right on the graph) it takes 4 times as long for nodes to transmit their share of A and B, and 8 times as many computation operations need to be done. Since it is communication-bound, we are doing 8 times as much computation in 4 times as much time, so the curves slope up at the rate of doubling the TFLOPs for each doubling of matrix size. In the left-most part of the graph, particularly for larger grid sizes, the amount of data being communicated



(a) (b)

Figure 4·7: Effect of using more transceivers. (a) 4096 DSPs, 4 MGTs in each direction. Most problem sizes are communication-bound. (b) 4096 DSPs, 16 MGTs in each direction. Larger problems now compute-bound. In all cases the smallest problems are latency-bound.



(a) (b)

Figure 4·8: These two charts consider using fewer DSP units. (a) 256 DSPs, 4 MGTs in each direction. (b) 256 DSPs, 16 MGTs in each direction.

is relatively small. The communication time ends up being dominated by the initial setup phase and by the communication latency. Thus the slope is steeper in that region. It is more prominent in the larger grids because these require a greater number of steps for the setup phase, and because with a larger number of FPGAs, each FPGA's share of the data is so small.

Moving to the far corner of the design space, in figure 4·7 (b) we have a much smaller 16 × 16 systolic array and a much larger communication bandwidth by using 16 MGTs for each direction on the grid. Almost all problem sizes are computation-limited, and plateau at a TFLOPs/sec level that corresponds to all MGTs on all FGPAs working full-time.

4.1.10 Comparison to Rival Architectures

Table 4.1 compares these results to a conventional HPC system and to a single node with GPU.

Table 4.1: MMM Performance Comparison

System	Nodes	Matrix Size	GFLOPs	GFLOPs /node	Theor. Peak	Utilization
Multi-FPGA 4K DSPs	1024	32768	2.38×10^6	2324	9200	25%
Multi-FPGA scaled	1024	32768	13000	12.7	13.6	93%
Multi-FPGA scaled	4096	32768	51900	12.7	13.6	93%
Blue Gene/P	1024	30000	8800	8.6	13.6	63%
Blue Gene/P	4096	30000	28700	7.0	13.6	51%
GeForce GTX 1080	1	10304	7500	7500	8300	90%

We compare the 1024- and 4096-node results from section 4.1.9 to the comparably-sized results from (Schatz et al., 2016). In that work the authors use an IBM Blue Gene/P system, citing a per-node theoretical peak computing rate of 13.6 GFLOPs/sec and networking bandwidth of 2.55 GB/sec per link.

Note that our results for Stratix 10 2800 FPGAs only utilize 25% of the manufacturer's claimed peak GFLOPs. This is primarily due to the conservative assumptions of section 4.1.6 : using a 300 MHz clock instead of 800 MHz; not using all of the DSP units. It would likely be possible to get much closer to the theoretical peak.

For easier comparison to our proposal, the lines in the table labeled "Multi-FPGA scaled" represent our FPGA design but using only 16 DSPs and 4 MGT links per FPGA, with clocks adjusted so that the computing rate of the DSPs exactly matches the Blue Gene/P node ratings. The table shows the results of that paper when using the Stationary type C version of eSUMMA-3D, a version of the Scalable Universal Matrix Multiply Algorithm that they optimized for use on the 3-D grid topology of the Blue Gene/P.

The results show that performance is comparable when the FPGA nodes' compute and communication bandwidth are adjusted to match the Blue Gene/P, and also show the performance gain that should be possible with our design.

The table also shows results from (Anders and Chrzęszczyk, 2017), an Nvidia article presenting source code for linear algebra operations using CUDA, along with some benchmark results on a GeForce GTX 1080. That GPU has a theoretical throughput of 8.3 TFLOPs/sec; the authors achieve about 90% of this on a dense matrix multiply with $N = 10304$.

4.2 Sparse Matrix Multiplication and Transpose

Large graphs (possibly directed, and with edge weights) are most naturally stored in sparse matrix form. Sparse matrix storage formats vary widely, see for example (Kestur et al., 2012); most can be characterized as compressed rows (or columns) possibly including non-compressed blocks of data in areas where the matrix is locally dense, and formatted to facilitate practical stream decompression. For this discussion we will assume that we wish to multiply two matrices A and B that are represented as an explicit list of tuples (i, j, A_{ij}) representing all nonzero elements.

For our purposes we are interested in exploring the problem of handling datasets that are too large to fit on a single node, or where it is desirable to use multiple nodes for gains in computing performance (in a multiplication or on other subsequent operations such as applying a nonlinear weighting function to each graph edge). Therefore we will start by assuming that the data for matrices A and B are already distributed fairly evenly among P nodes. Each node initially transfers its share of the A and B data to its FPGA and asks the network to return its share of the product matrix $C = A \times B$.

The actual multiplication requires matching up all the elements of A in each column i with the elements of B in the ith row. To get all such data localized onto the same node, we want to transpose B and keep the data partitioned so that all data from A and B with the needed matching indexes end up on the same node. Therefore, the sparse matrix multiply solution depends on a solution for sparse matrix transpose.

4.2.1 Sparse Matrix Transpose

For the subproblem of transposing an $n \times m$ matrix B, the data consist of tuples (i, j, B_{ij}) representing all nonzero elements initially sorted in row-major order. It is distributed roughly evenly among P nodes of the system: each node contains a set of consecutive members of the sorted list. The task is to rearrange the data so that it is sorted in column-major order.

In the initial ordering, the data are partitioned by P-quantiles according to the row-major ordering. There are $P - 1$ quantile points each expressible as a pair of values (i_k, j_k). These values could be used to quickly determine which node holds a given datum. For example, the node with rank 3 holds the 4^{th} quantile, which would include all data from (i_3, j_3) up to but not including (i_4, j_4).

For the desired reordering, there are a different set of P-quantiles corresponding to the partitioning of the set in column-major order. If these $P-1$ tuples were known, each node could easily determine where each of its data needs to be sent using $P-1$ local comparisons performed in parallel. The primary task is to determine these quantiles.

We also want the A dataset, already sorted and not needed any rearrangement, to be partitioned according to the same quantiles. In other words, for the purpose of locating the P-quantiles, the distribution of both A and B data need to be considered together. Hereafter it will be assumed that is the case.

4.2.2 Balanced Bucket Sort With Radix Search

Given unordered data and desiring to know the ith P-quantile, we suggest an algorithm analogous to binary search. Start with a range $[low..high]$ that covers all data values. Choose a test value *mid* that is halfway between *low* and *high*. Perform a single scan through all the data, counting how many are less than *mid*. If that count is less than i/P times the total number of data, set the range to $[mid, high]$, otherwise set the range to $[low, mid]$; then repeat the process. Continue until the endpoints of the range are identical; then they are equal to the ith P-quantile.

This process requires as many steps (full scans of all data) as the number of bits of precision in the value that determines the ordering. In the case of transposing a sparse matrix, the number of bits required to represent the row and column indexes is $\lceil \log_2(n+m) \rceil$ for a sparse matrix of dimensions $m \times n$. It would require this many complete passes through the dataset for all P-quantiles to be found.

Each of these passes would consist of the P nodes streaming the entire dataset to each other, in a ring communication pattern. As described in section 3.2.5, with at least 8 MGTs per link, this is greater bandwidth than can be achieved on a single node through a DDR controller.

In a *radix search*, the same algorithm is used, except that some larger number of comparators are used to simultaneously compare the data to each of a larger number of candidate *mid* values. After all data have been counted, the range is set to the two compare-points whose counts straddle the desired quantile point. If the number of comparators employed is $r - 1$ (where r is the radix, originally 2 in the case of binary search), then the number of passes through the whole data set reduces to $\lceil \log_r (n + m) \rceil$.

The overall algorithm is a distributed bucket sort, but maintaining balanced bucket sizes. The part just described, determining the ranges for each bucket, has algorithmic complexity $O(nlogn)$ where the logarithm is to base r.

To utilize a 2D grid rather than a simple ring, consider this modification of the radix search for P nodes arranged in a $2 \times P/2$ 2-D torus. Each row holds half of the full A and B data, and the communication occurs only east-west as before. The two nodes in a column share the same $[low, mid_1, mid_2, ..., high]$ comparison values, and both count the data falling in each bin; then combine their totals via a short communication to each other, to determine the new smaller $[low, high]$ ranges for each quantile. This allows the complete determination of all N-quantile points to be done twice as quickly, because the same number of MGT links are being used in the east-west direction, but each only has to carry half as much data during each step of the radix search. For more performance increase the number of rows, until the additional latency of the north-south synchronization approaches the saved time to send all data east-west.

Once the N-quantiles are known, transmit all data around the ring one more time, and each node keeps its share of the data, being those data that fall in its bucket. In the $2 \times P/2$ modification of the previous paragraph, each pair of nodes must work in pairs, using the north-south links to exchange data destined for each other's buckets

The design can include a little extra memory, so that each of the P nodes can hold a bit more than their $1/P$ share of the dataset; in that case it will be possible to begin the local sorting of the buckets before the P-quantiles have been determined precisely. Each P-quantile will definitely be somewhere within its $[low, high]$ range at the end of each step of the radix search algorithm, and so the entire dataset needed by node i will be somewhere between low_i and $high_{i+1}$. Once this range is small enough so that the data in that range fit in local BRAM, that data can be sorted locally. Each node will have extra data that can be discarded once the exact quantiles are known.

A local sorting hardware design is a bit beyond the scope of this thesis, but there are many results applicable to our situation. Refer to (Chen et al., 2015) and (Matai et al., 2016) for examples.

Once matrix B has been transposed to column-major order and its elements completely sorted, the multiplication $C = A \times B$ can be done in a single pass by each node without further communication and in a highly parallel manner, with the result C in row-major order.

4.2.3 Performance Comparison to a Single Node

The sparse transpose on a single node will be a sorting problem requiring $O(nlogn)$ memory accesses limited by the bandwidth of the memory holding the data, and perhaps about half of the sorting algorithm's passes will have enough spatial locality to use the cache efficiently. As described here, a multi-node sparse transpose requires $O(logn + logm)$ passes through the entire N data limited by the bandwidth of the communication along one edge of the network topology graph. As described in section 3.2.5, if each network link is using 8 or 16 MGTs, this bandwidth will compare favorably with the hard DDR4 controller on an FPGA. Using a rectangular grid instead of a ring, the multi-node implementation has a great bandwidth advantage.

Though we have not explored this algorithm in sufficient detail to produce accurate performance comparisons, we suggest that it is an area worthy of further research.

Chapter 5

Summary, and Future Considerations

Given the limitations and constraints of communication latency and bandwidth on high-performance computing, we have identified a need for a more robust and comprehensive Compute in Network capability. We have described a design model for HPC incorporating computing logic at every node in a way that provides for computation tightly coupled with switching, and allowing interior nodes to altruistically perform computations that might otherwise need to occur after input data are delivered to leaf nodes. Central to our model is the use of the FPGA as the provider of all communication and switching, so that the amount of hardware devoted to altruistic computation can be varied as is suitable for each application.

We show how a few broadly useful types of distributed computation can be efficiently executed by a gridlike network of FPGAs with greatly enhanced interconnection links.

5.1 Impact of Future Products

We now note that planned future products such as Stratix 10 MX (Intel Corporation, 2017a) include a large amount of DRAM, using multiple dies within a single package. These provide a much higher memory bandwidth than can be achieved with a current-generation FPGA with a single DDR memory channel. Such a product would be a good choice in the type of design we are proposing, a D" system with each node including a standard CPU. The greater local memory in the FPGA would allow larger datasets to be processed in the ways we described and broaden the range of of applications suitable to optimization via AC and CiN.

There have been multiple reports, e.g. (Williams, 2016), of future products that integrate a CPU and an FPGA within a package. If and when such products are available, the effect will be similar to the FPGA with integrated DRAM. The closer connection to CPU will make it even easier to offload some computing functions to the network processor. The core advantage of our Compute in Network thesis remains: the ability of nodes to altruistically compute on data that is in transit between two other (source and destination) nodes.

In section 4.1.10 we showed a single-node result using a single Nvidia GPU with peak TFLOPs about the same as our FPGA. Nvidia GPUs can be connected with a scalable interconnect called NVLink; at this writing it is possible to connect up to 16 GPUs in a single-node server configuration with each GPU having up to 300 GB/sec bandwidth through the links (Nvidia, 2018). For some applications, this or its likely successors will rival or surpass many HPC alternatives.

5.2 Areas for Future Research

In this thesis we have provided only outlines of designs and estimates of relative performance. It is appropriate to develop these ideas further by creating actual FPGA designs and testing them in simulation.

If such tests yield promising results, it would then be appropriate to perform real tests on FPGA hardware with multiple nodes and many high-speed communication links over cable lengths like those in large-scale HPC systems.

Further development of these ideas should occur in parallel, and when possible exchange design decisions with, ongoing development of the next version of the MPI standard.

References

Agron Design (2013). Argon Design announces groundbreaking results for high performance trading with FPGA and x86 technologies. *Industry press release, www.thetradingmesh.com/pg/newsfeeds/argon/item/124341*.

Altera (2017). Intel Stratix 10 clocking and PLL user guide. *www.altera.com/en_US/pdfs/literature/hb/stratix-10/ug-s10-clkpll.pdf*.

Anders, J. and Chrzęszczyk, A. (2017). Matrix computations on the GPU – CUBLAS, CUSOLVER and MAGMA by example. *developer.nvidia.com/sites/default/files/akamai/cuda/files/Misc/mygpu.pdf*.

Arista Networks (2013). 7124FX application switch data sheet. *www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf*.

Baxter, R., Booth, S., Bull, M., Cawood, G., Perry, J., Parsons, M., Simpson, A., Trew, A., McCormick, A., Smart, G., et al. (2008). Maxwell–a 64 FPGA supercomputer. *Engineering Letters*, 16(3).

Caulfield, A., Chung, E., Putnam, A., et al. (2016). A cloud-scale acceleration architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 13–24. IEEE/ACM.

Chang., K. K. (2017). *Understanding and Improving the Latency of DRAM-Based Memory Systems*. Ph.D. dissertation, Carnegie Mellon University.

Chen, R., Siriyal, S., and Prasanna, V. (2015). Energy and memory efficient mapping of bitonic sorting on fpga. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 240–249. ACM.

Chiu, M. and Herbordt, M. (2009). Efficient filtering for molecular dynamics simulations. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*.

Chiu, M. and Herbordt, M. (2010). Molecular dynamics simulations on high performance reconfigurable computing systems. *ACM Transactions on Reconfigurable Technology and Systems*, 3(4):1–37.

Chiu, M., Herbordt, M., and Langhammer, M. (2008). Performance potential of molecular dynamics simulations on high performance reconfigurable computing systems. In *Proceedings High Performance Reconfigurable Technology and Applications*.

Chiu, M., Khan, M., and Herbordt, M. (2011). Efficient calculation of pairwise nonbonded forces. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*.

Conti, A., VanCourt, T., and Herbordt, M. (2004). Flexible FPGA acceleration of dynamic programming string processing. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*.

Dally, W. and Towles, B. (2004). *Principles and Practices of Interconnection Networks*. Elsevier.

DOE (2011). *CoDEx: CoDesign for Exascale*. United States Department of Energy Offic of Science, Advanced Scientific Computing Research.

Gara, A., Blumrich, M. A., Chen, D., Chiu, G.-T., Coteus, P., Giampapa, M. E., Haring, R. A., Heidelberger, P., Hoenicke, D., Kopcsay, G. V., et al. (2005). Overview of the Blue Gene/L system architecture. *IBM Journal of research and development*, 49(2.3):195–212.

Geijn, R. A. V. D. and Watts., J. (1997). SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency—Practice and Experience*, 9(4):255–274.

Geng, T., Wang, T., Sanaullah, A., Yang, C., Xuy, R., Patel, R., and Herbordt, M. (2018). Multicore versus fpga in the acceleration of discrete molecular dynamics. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*.

George, A., Herbordt, M., Lam, H., Lawande, A., Sheng, J., and Yang, C. (2016). Novo-G#: A Community Resource for Exploring Large-Scale Reconfigurable Computing Through Direct and Programmable Interconnects. In *Proceedings of the IEEE High Performance Extreme Computing Conference*.

Gokhale, M., Lloyd, S., and Macaraeg, C. (2015). Hybrid memory cube performance characterization on data-centric workloads. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, page 7. ACM.

Google Cloud (2017). An in-depth look at Google's first tensor processing unit (TPU). *cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu*.

Gropp, W., Lusk, E., and Skjellum, A. (1998). *Using MPI: Portable Parallel Programming with the Message-passing Interface, Volume 1, The MPI Core, second edition.* The MIT Press.

Gu, Y. and Herbordt, M. (2007). FPGA-based multigrid computations for molecular dynamics simulations. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*, pages 117–126.

Gu, Y., VanCourt, T., and Herbordt, M. (2005). Accelerating molecular dynamics simulations with configurable circuits. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications.*

Gu, Y., VanCourt, T., and Herbordt, M. (2008). Explicit design of FPGA-based coprocessors for short-range force computation in molecular dynamics simulations. *Parallel Computing*, 34(4-5):261–271.

Gunnels, J., Lin, C., Morrow, G., and van de Geijn, R. (1996). Analysis of a class of parallel matrix multiplication algorithms. *IPPS (IEEE Parallel Processing Symposium) 1998, as "A flexible class of parallel matrix multiplication algorithms".*

Herbordt, M. (2018). Towards production HPC with FPGA-centric clouds and clusters. In *SIAM conference on Parallel Processing in Scientific Computing.*

Herbordt, M., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., and Chiu, M. (2008). Computing models for FPGA-based accelerators with case studies in molecular modeling. *Computing in Science and Engineering*, 10(6):35–45.

Herbordt, M., Khan, M., and Dean, T. (2009). Parallel discrete event simulation of molecular dynamics through event-based decomposition. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 129–136.

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2006). Single pass, BLAST-like, approximate string matching on FPGAs. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines.*

Herbordt, M., Model, J., Sukhwani, B., Gu, Y., and VanCourt, T. (2007a). Single pass streaming BLAST on FPGAs. *Parallel Computing*, 33(10-11):741–756.

Herbordt, M., Olin, K., and Le, H. (1999). Design trade-offs of low-cost multicomputer networks. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 25–34.

Herbordt, M., VanCourt, T., Gu, Y., Sukhwani, B., Conti, A., Model, J., and DiSabello, D. (2007b). Achieving high performance with FPGA-based computing. *IEEE Computer*, 40(3):42–49.

Hillis, W. D. (1984). The connection machine: A computer architecture based on cellular automata. *Physica D: Nonlinear Phenomena*, 10(1-2):213–228.

Humphries, B., Zhang, H., Sheng, J., Landaverde, R., and Herbordt, M. (2014). 3D FFT on a Single FPGA. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines.*

Intel Corporation (2015a). Hyper-Pipelining for Stratix 10 designs. *www.altera.com/en_US/pdfs/literature/an/an715.pdf*.

Intel Corporation (2015b). Intel Xeon processor E7-4830 v3. *ark.intel.com/products/84678*.

Intel Corporation (2016). Designing for Stratix 10 devices with power in mind. *www.altera.com/en_US/pdfs/literature/an/an767.pdf*.

Intel Corporation (2017a). Intel Stratix 10 MX devices solve the memory bandwidth challenge. *www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01264-stratix10mx-devices-solve-memory-bandwidth-challenge.pdf*.

Intel Corporation (2017b). Intel Stratix 10 variable precision DSP blocks user guide. *www.altera.com/documentation/kly1436148709581.html*.

Intel Corporation (2017c). Intel Xeon Silver 4116 processor. *ark.intel.com/products/120481*.

Intel Corporation (2018). External memory interface spec estimator. *www.altera.com/support/support-resources/support-centers/external-memory-interfaces-support/emif.html*.

Kestur, S., Davis, J. D., and Chung, E. S. (2012). Towards a universal FPGA matrix-vector multiplication architecture. In *IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16. IEEE.

Khan, M. and Herbordt, M. (2011). Parallel discrete event simulation of molecular dynamics with speculation and in-order commitment. *Journal of Computational Physics*, 230(17):6563–6582.

Khan, M. and Herbordt, M. (2012). Communication requirements for FPGA-centric molecular dynamics. In *Symposium on Application Accelerators for High Performance Computing.*

Kogge, P. M. (1994). EXECUBE–a new architecture for scaleable MPPs. In *ICPP (International Conference on Parallel Processing), 1994.*, volume 1, pages 77–84. IEEE.

Li, J., Skjellum, A., and Falgout., R. D. (1993). A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Center for Computational Sciences and Engineering 50*, page 316.

LinuxGizmos (2017). ARM/FPGA module runs Debian on Arria 10 SoC. *linuxgizmos.com/arm-fpga-module-runs-debian-on-arria-10-soc/*.

Liu, Y., Sheng, J., and Herbordt, M. (2016). A Hardware Prototype for In-Brain Neural Spike-Sorting. In *Proceedings of the IEEE High Performance Extreme Computing Conference*.

Mahram, A. and Herbordt, M. (2012). FMSA: FPGA-Accelerated ClustalW-Based Multiple Sequence Alignment through Pipelined Prefiltering. In *Proceedings of the 20th International Symposium on Field Programmable Custom Computing Machines*, pages 177–183.

Mahram, A. and Herbordt, M. (2016). NCBI BLASTP on High Performance Reconfigurable Computing Systems. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):6.1–6.20.

Matai, J., Richmond, D., Lee, D., Blair, Z., Wu, Q., Abazari, A., and Kastner, R. (2016). Resolve: Generation of high-performance sorting architectures from high-level synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 195–204. ACM.

NSF (2006). *Simulation-Based Engineering Science*. NSF (National Science Foundation) Blue Ribbon Panel on Simulation-Based Engineering Science.

Nvidia (2018). NVLink Fabric—a faster, more scalable interconnect. *www.nvidia.com/en-us/data-center/nvlink/*.

Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., and Yelick, K. (1997). A case for intelligent RAM. *IEEE Micro*, 17(2):34–44.

Pawlowski, J. T. (2011). Hybrid memory cube (HMC). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24. IEEE.

Pres. (2005). *Computational Science: Ensuring America's Competitiveness*. President's Information Technology Advisory Committee, National Coordination Office for Information Technology Research and Development, www.nitrd.gov.

Putnam, A. et al. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3).

Salapura, V., Bickford, R., Blumrich, M., Bright, A. A., Chen, D., Coteus, P., Gara, A., Giampapa, M., Gschwind, M., Gupta, M., et al. (2005). Power and performance optimization at the system level. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 125–132. ACM.

Sanaullah, A. and Herbordt, M. (2018). FPGA HPC using OpenCL: Case Study in 3D FFT. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies.*

Sanaullah, A., Khoshparvar, A., and Herbordt, M. (2016a). FPGA-Accelerated Particle-Grid Mapping. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines.*

Sanaullah, A., Lewis, K., and Herbordt, M. (2016b). Accelerated Particle-Grid Mapping. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis – Supercomputing.*

Sanaullah, A., Yang, C., Alexeev, Y., Yoshii, K., and Herbordt, M. (2018). Real-Time Data Analysis for Medical Diagnosis using FPGA Accelerated Neural Networks. *BMC Bioinformatics*, In Press.

Schatz, M. D., Van de Geijn, R. A., and Poulson, J. (2016). Parallel matrix multiplication: A systematic journey. *SIAM Journal on Scientific Computing*, 38(6):C748–C781.

Sheng, J. (2017). *High Performance Communication on FPGA-Centric Clusters.* PhD thesis, Department of Electrical and Computer Engineering, Boston University.

Sheng, J., Humphries, B., Zhang, H., and Herbordt, M. (2014). Design of 3D FFTs with FPGA Clusters. In *Proceedings of the IEEE High Performance Extreme Computing Conference.*

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2015a). Hardware-Efficient Compressed Sensing Encoder Designs for WBSNs. In *Proceedings of the IEEE High Performance Extreme Computing Conference.*

Sheng, J., Xiong, Q., Yang, C., and Herbordt, M. (2016a). Collective Communication on FPGA Clusters with Static Scheduling. *Computer Architecture News*, 44(4).

Sheng, J., Yang, C., Caulfield, A., Papamichael, M., and Herbordt, M. (2017). HPC on FPGA Clouds: 3D FFTs and Implications for Molecular Dynamics. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications.*

Sheng, J., Yang, C., and Herbordt, M. (2015b). Towards Low-Latency Communication on FPGA Clusters with 3D FFT Case Study. In *Proceedings of the International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies.*

Sheng, J., Yang, C., and Herbordt, M. (2016b). Application-Aware Collective Communication on FPGA Clusters. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*.

Sheng, J., Yang, C., and Herbordt, M. (2018). High Performance Dynamic Communication on Reconfigurable Clusters. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*.

Skjellum, A. (2017). personal communication.

Song, Y. and Parihar, R. (2012). DRAM memory controller and optimizations (class project report, University of Rochester). *www.hajim.rochester.edu/ece/parihar/pres/Pres_DRAM-Scheduling.pdf*.

Sukhwani, B. and Herbordt, M. (2008). Acceleration of a Production Rigid Molecule Docking Code. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*, pages 341–346.

Sukhwani, B. and Herbordt, M. (2009a). Accelerating CHARMM Energy Minimization Using Graphics Processors. In *Proceedings of the Symposium on Application Accelerators in High Performance Computing*.

Sukhwani, B. and Herbordt, M. (2009b). FPGA-Acceleration of CHARMM Energy Minimization. In *Proceedings of the Third International Workshop on High Performance Reconfigurable Computing Technology and Applications*. ACM.

Sukhwani, B. and Herbordt, M. (2010). FPGA Acceleration of Rigid Molecule Docking Codes. *IET Computers and Digital Techniques*, 4(3):184–195.

Sukhwani, B. and Herbordt, M. (2014). Increasing Parallelism and Reducing Thread Contentions in Mapping Localized N-body Simulations to GPUs. In Kindratenko, V., editor, *Numerical Computations with GPUs*. Springer Verlag.

VanCourt, T., Gu, Y., and Herbordt, M. (2004). FPGA acceleration of rigid molecule interactions. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*.

VanCourt, T. and Herbordt, M. (2004). Families of FPGA-based algorithms for approximate string matching. In *Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors*, pages 354–364.

VanCourt, T. and Herbordt, M. (2005a). LAMP: A tool suite for families of FPGA-based application accelerators. In *Proceedings of the IEEE Conference on Field Programmable Logic and Applications*.

VanCourt, T. and Herbordt, M. (2005b). Three dimensional template correlation: Object recognition in 3D voxel data. In *IEEE International Workshop on Computer Architectures for Machine Perception*, pages 153–158.

VanCourt, T. and Herbordt, M. (2006). Rigid molecule docking: FPGA reconfiguration for alternative force laws. *Journal on Applied Signal Processing*, v2006:1–10.

VanCourt, T. and Herbordt, M. (2007). Families of FPGA-based accelerators for approximate string matching. *Microprocessors and Microsystems*, 31(2):135–145.

VanCourt, T. and Herbordt, M. (2009). Elements of high performance reconfigurable computing. In Zelkowitz, M., editor, *Advances in Computers*, volume v75, pages 113–157. Elsevier.

Williams, C. (2016). Here's what an Intel Broadwell Xeon with a built-in FPGA looks like. *www.theregister.co.uk/2016/03/14/intel_xeon_fpga/*.

Xiong, Q. and Herbordt, M. (2017). Bonded Force Computations on FPGAs. In *Proceedings of the IEEE International Symposium on Field Programmable Custom Computing Machines*.

CURRICULUM VITAE

[redacted]

[redacted]