AFRL-RY-WP-TR-2021-0007

DARPA SOFTWARE DEFINED HARDWARE

Jonathan P. Skeans
University of Dayton Research Institute

AUGUST 2021
Final Report

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

See additional restrictions described on inside pages

STINFO COPY

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.

Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2021-0007 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

//Signature//

MARC P. HOFFMAN
Program Manager
Sensors Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

TIMOTHY R. JOHNSON
Sensors Subsystems Branch
Aerospace Components & Subsystems Division

//Signature//

LESTER C. LONG Lt Col. USAF
Deputy
Aerospace Components & Subsystems Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
August 2021	Final	3 December 2018 – 3 December 2020

4. TITLE AND SUBTITLE
DARPA SOFTWARE DEFINED HARDWARE

5a. CONTRACT NUMBER
FA8650-19-2-7906

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62716E

6. AUTHOR(S)
Jonathan P. Skeans

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
Y1YT

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Dayton Research Institute
300 College Park Avenue
Dayton, OH 45435

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency
(DARPA/MTO)
675 North Randolph Street
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDR

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2021-0007

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with The Under Secretary of Defense memorandum dated 24 May 2010 and AFRL/DSO policy clarification email dated 13 January 2020. This report is available to the general public, including foreign nationals.. This material is based on research sponsored by the Air Force Research Lab (AFRL) and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-19-2-7906. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Labs (AFRL), the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government. Report contains color.

14. ABSTRACT
To benchmark the capabilities of novel electronic processing devices being developed by DARPA, six common algorithms, or kernels, were selected to be used for comparison purposes. The kernels include Fast Fourier Transforms, Dense and Sparse Matrix Multiplication, Two Dimensional Convolution, Dijkstra's Algorithm, and the Auction Algorithm. The processing performance of these kernels on new devices, in terms of speed and power requirements, are to be compared against the performance on modern Application Specific Integrated Circuits (ASICs). Since ASICs for these specific kernels are not equally available, Field Programmable Gate Arrays (FPGAs) are used as proxies. For each kernel, an operations per watt value is derived when implemented on FPGAs, from which the metric for ASIC performance is estimated.

15. SUBJECT TERMS
High Performance Computing, Edge Processing, Field Programmable Gate Array (FPGA)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Marc Hoffman
Unclassified	Unclassified	Unclassified	SAR	31	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Table of Contents

Section **Page**

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

List of Figures

Figure **Page**

List of Tables

Table **Page**

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

1 INTRODUCTION

Recent developments in graph analytics and data-intensive processing have demonstrated that conventional central processing unit (CPU) and graphics processing unit (GPU) processing architectures are ill-suited for efficient performance. This mismatch of technologies has motivated the Defense Advanced Research Projects Agency (DARPA) software defined hardware (SDH) program to develop specialized reconfigurable hardware processing architectures with the goal of building runtime-reconfigurable hardware and software that enables near ASIC performance without sacrificing programmability for data-intensive algorithms. A measure of merit for DARPA SDH is to compare the specialized hardware solutions with baseline CPU, GPU, and field programmable gate array (FPGA) implementations, providing an FPGA proxy of application specific integrated circuit (ASIC) implementations of fundamental and representative applications code. The amount of labor required to design highly optimized FPGA implementations establishes a risk for the program. University of Dayton Research Institute (UDRI) has shown unique capability in the areas of scalable computing and abstracted FPGA development and is well suited to employ emerging techniques for FPGA development to provide these baselines in a cost effective and low-risk manner.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2 STATEMENT OF WORK

The proposed study seeks to leverage emerging higher-level languages to develop FPGA baseline approximations, with FPGA implementations serving as proxies for ASIC implementations, for six graph processing/graph analytics algorithms in one year to support the DARPA SDH program for architecture solution evaluation and analysis. The SDH workflow kernels to be considered for implementation are shown below in Table 1.

Table 1. Proposed SDH Workflow Kernels

SDH Workflow Kernel	Referenced in SDH Workflows
Fast Fourier Transform	Used in many workflows, especially signal processing and differential equations
General matrix multiply (GEMM)	Speech-to-Text, Dependency Parsing, Randomized SVD, Node Classification, Image Classification, Proximal Policy Optimization, Mel-frequency Cepstrum Coefficients, Recommender System, Seeded Graph Matching
General sparse matrix multiply (SpGEMM)	Speech-to-Text, Role Prediction, Naïve Bayes SGD Classifier, Recommender System
Graph Traversal (Random Walk/ Graph Search)	Naïve Bayes SGD Classifier, Assignment Problem, DBSACN, Vehicle Routing, Graph Search
Stochastic Gradient Descent (Linear SGD Classifier)	
Convolution (2D convolutions)	Video Segmentation, Speech-to-Text, Optical Flow, Image Segmentation, Image Classification, MiniGo, Proximal Policy Optimization
QR Decomposition	Randomized SVD
Hash Table	Naïve Bayes SGD Classifier, Vertex Nomination, Vehicle Routing, Random Walk

2.1 FFT

The first kernel being developed is the FFT. As mentioned in the last monthly status report, Intel has what's known as a hybrid FFT core which means that it is a parallel implementation. The parallelism is achieved on a single FFT as opposed to processing multiple FFTs at once. FFT cores were implemented that supported FFT sizes from 1K to 64K. Using Quartus's integrated power analyzer tool, the power estimates were determined. These estimates varied depending on the level of parallelism, FFT size, and core clock. The best performance estimate achieved up to the time of reporting is 347 GFLOPS/W running at 750 MHz on a Stratix 10.

2.2 Dense Matrix Multiplication

As mentioned in last report, the issue with the systolic array design that was being implemented was scaling to the required sizes dictated by the SDH workflows. Each processing element consumes a single high performance, low latency hardened DSP block which means that the design can only be as large as the number of digital signal processing (DSP) blocks in the FPGA. The largest Stratix 10 FPGA will not accommodate the data sizes that are required. Literature searches were done to find a more optimal design.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

As described in [1], an energy efficient FPGA matrix multiplication is presented. This design presents several advantages over the pure systolic array implementation. An on-chip memory design is shown in Figure 1

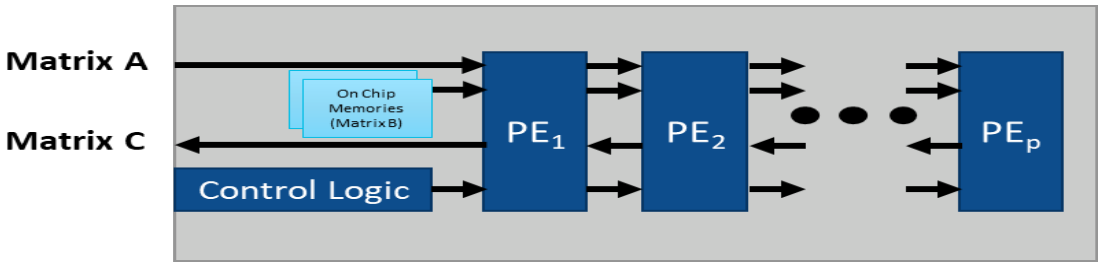

Figure 1: On-chip Memory Design

This design consists of a linear systolic array which minimizes the number of long interconnects and can lead to improved energy efficiency. Each processing element only communicates with its nearest neighbor. This design also leverages both parallel and pipelined processing. Parallel processing can be achieved by increasing the number of resources in each processing element which leads to higher throughput. The systolic designed allows for pipelining which increases the resource utilization. NxN matrix multiplication requires only N hardened DSP floating point blocks as opposed to NxN DSP blocks for the previous design. This DSP reduction will allow the SDH data sizes to fit within a Stratix 10 devices. The GX2800 variant of the Stratix 10 consist of 5760 DSP blocks. Therefore theoretically, the largest matrix multiplication that can be done is 5760x5760. This does not consider the on-chip memory requirements for each processing element. A more detailed view of a processing element can be seen in Figure 2.

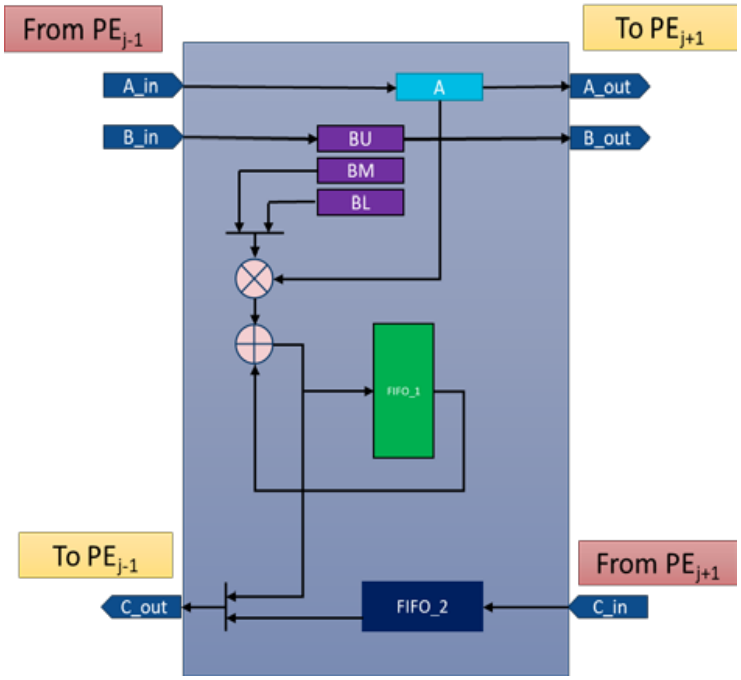

Figure 2: Processing Element Architecture

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Using the architecture in Figure 2, NxN matrix multiplication can be completed in n2+2n cycles with each processing element using 1 DSP. Each processing element also consist of 3 IO ports, 4 registers, and 2 FIFOs of N word capacity.

To improve throughput, more resources can be used for each processing element as shown in Figure 3 below.

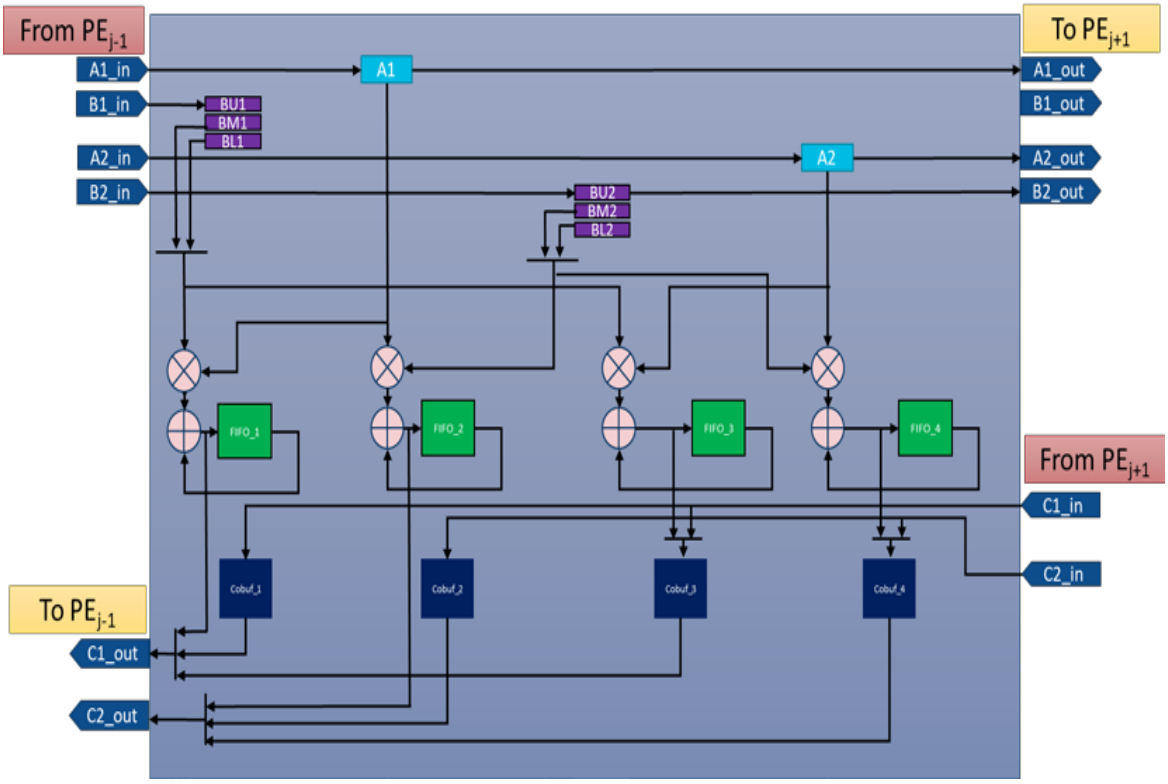

Figure 3: Matrix Multiply Parallel Architecture

The throughput is increased by a factor of r, the number of processing elements is reduced by a factor of r, while the number of DSPs for each processing element increases by a factor of r2. The simple core shown in Figure 2 initially did not scale very well in terms of Fmax.

Scaling the design reduced the Fmax considerably. Therefore, most of the effort was concentrated on modifying the architecture to improve scalability and achieving a stable Fmax rather than functionality. This involved including additional delay paths and registers. Additional effort would be needed to get the optimized architecture functionally correct.

An Fmax of 630 MHz was achieved targeting a Stratix 10 device at 30 Watts using the parallel architecture with r = 4.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2.3 2D Convolution

2D Convolution was implemented using Intel's DSP Builder tool. Early literature searches showed that depth-wise convolution can perform the standard convolution algorithm with significantly fewer operations as illustrated in Figure 4.

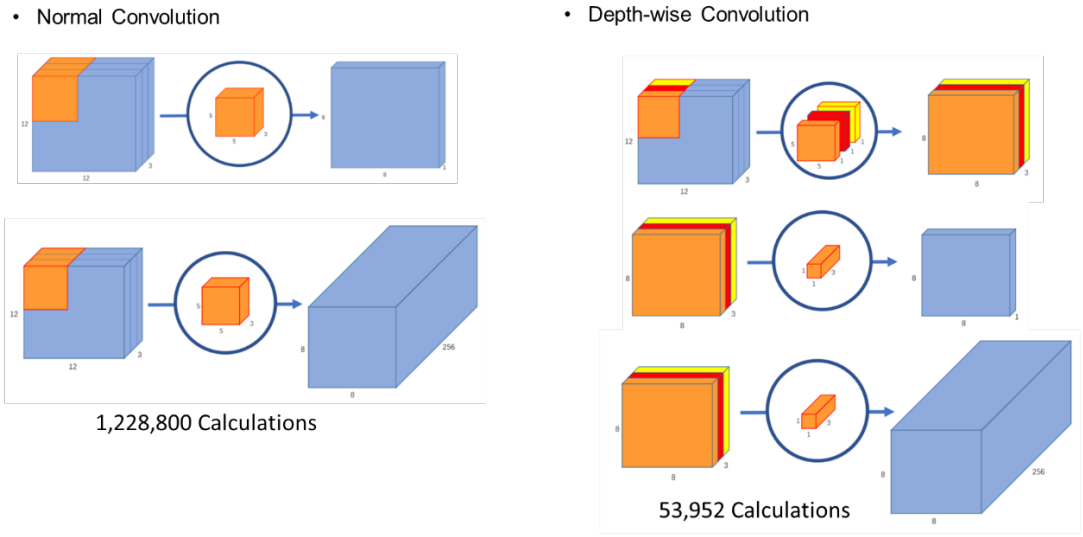

Figure 4: Standard Convolution vs Depth-Wise Convolution

The convolution algorithm was implemented as a systolic array as shown below Figure 5.

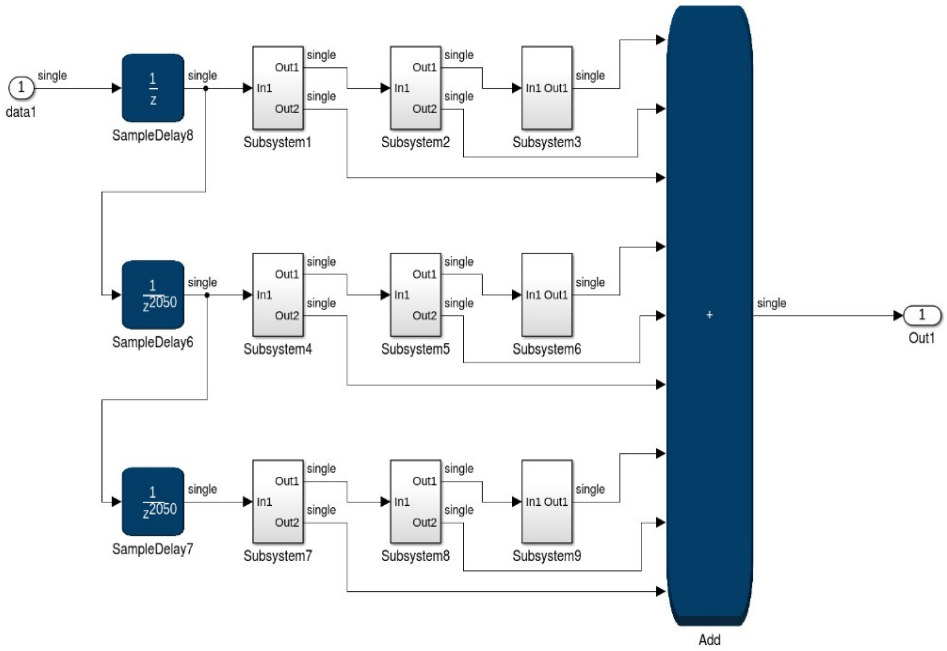

Figure 5: Systolic Array Design

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This design was very efficient in terms of Fmax. An Fmax of nearly 800 MHz was achieved. However the design is IO limited and only 4-5 convolutions cores exhaust the IO bandwidth for a standard Stratix 10 as shown in Figure 6.

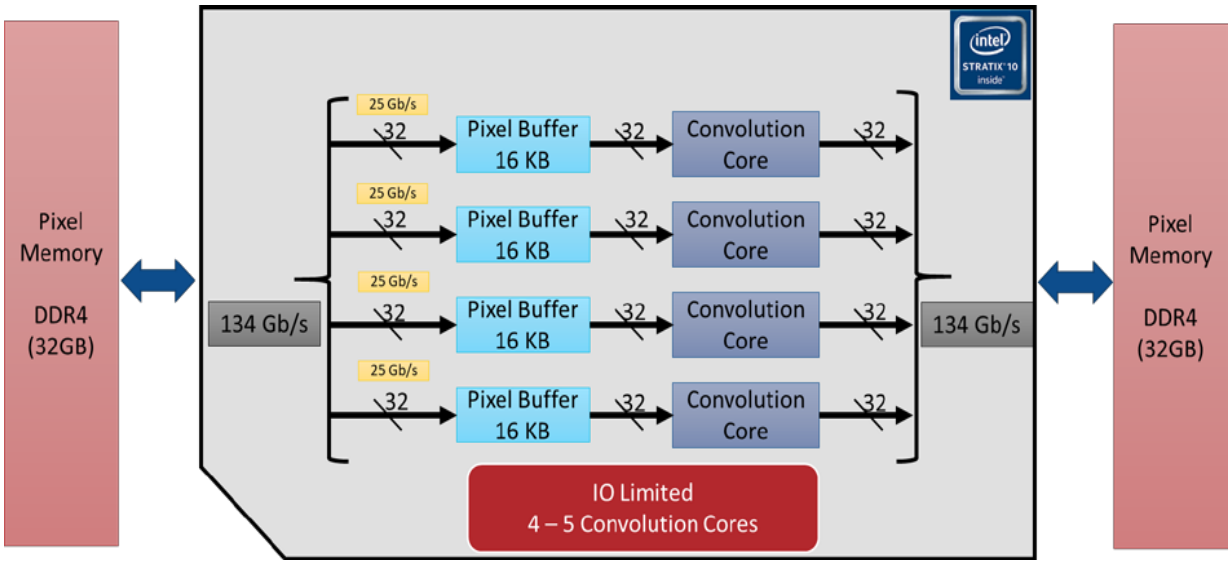

Figure 6: I/O Limited Convolution Design

To solve this problem, a different variant of Stratix 10 FPGAs was explored, the MX variant. The MX variant consists of up to 16 GB of high band-width memory which can solve the IO constraints as shown in Figure 7 below. Using this design, an estimated 64 convolution cores can fit on the FPGA.

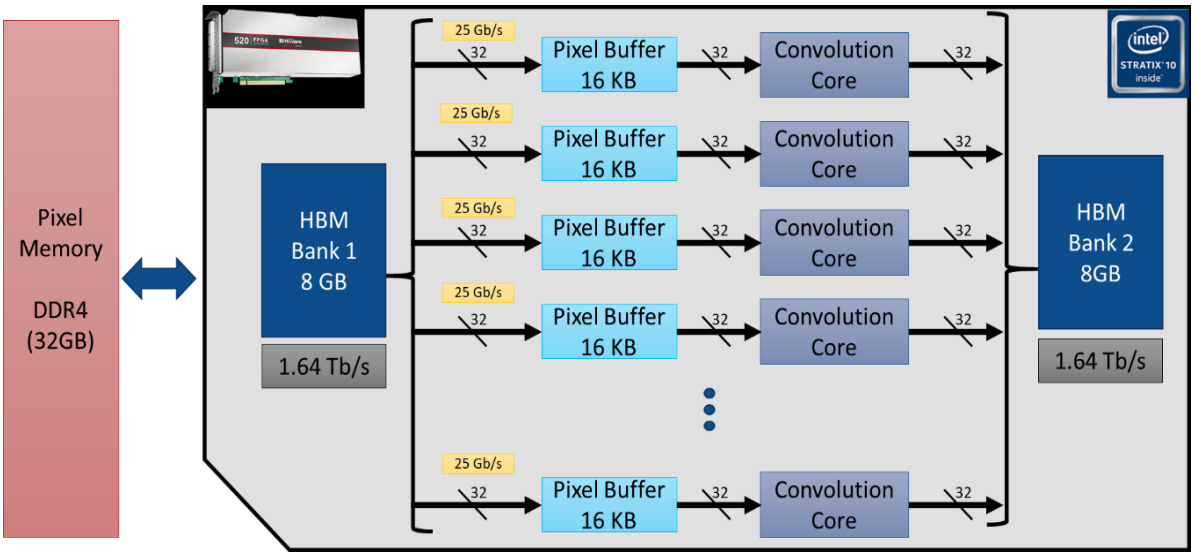

Figure 7: Convolution using Stratix 10 with High Band-Width Memory

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

2.4 Sparse Matrix Multiplication

Sparse matrix multiplication was implemented with some assumptions up front. The first being that the A matrix is in compressed sparse row (CSR) format. Second being that the size of the A matrix is (<1024, 10k-100k). Third, the B matrix is stored in memory because it usually doesn't not change. And last, the B matrix is of size (10k100k, 32-15). Sparse matrix multiplication is depicted below in Figure 8.

The number of cycles until complete is dependent upon the sparsity of the matrix.

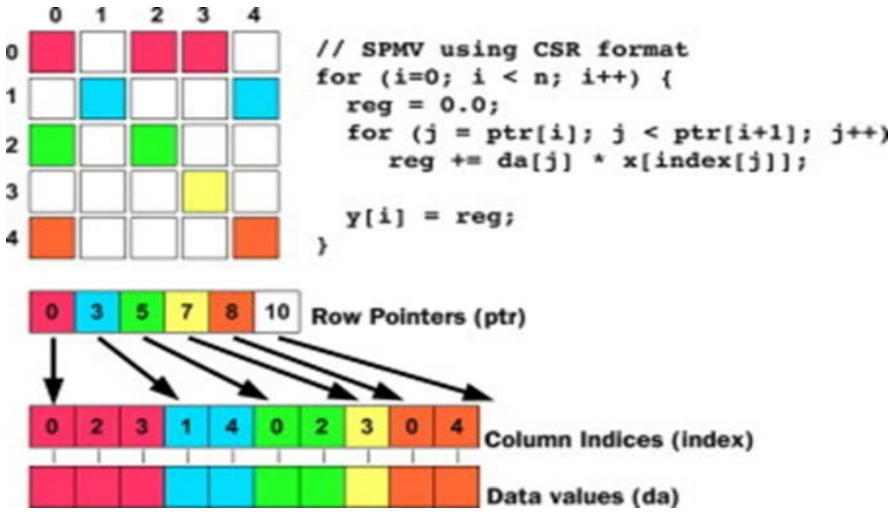

Figure 8: SPMV using CSR Format

The number of cycles until complete is dependent upon the sparsity of the matrix. If more non-zeros than rows, the latency is approximately 1.0005 the number of nonzeros. If more rows than non-zeros, latency is 2.5x the number of rows. The FPGA design that was implemented using DSP Builder is shown in Figure 9. The CSR formatted sparse matrix is ingested. The rows are processed as soon as the Row Processor subsystem is available. The output data is ordered correctly in the Build output subsystem.

$$A \times B = C$$

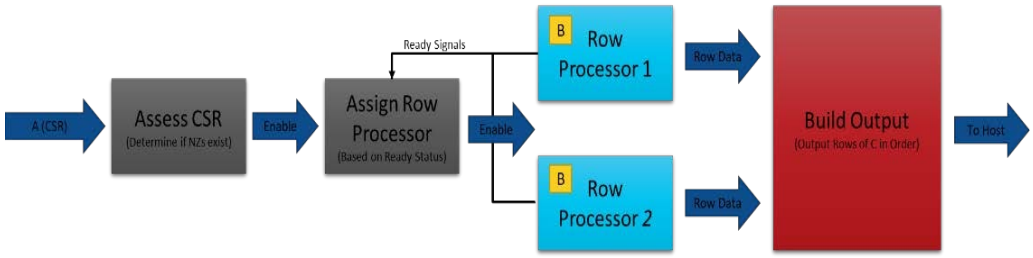

Figure 9: Sparse Matrix Functional Design

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

3 DIJKSTRA ALGORITHM

The key to parallelizing Dijkstra-like algorithms is eliminating the von Neumann bottleneck. In this research, we have achieved that by designing a Processor in Memory (PIM) random-access memory (RAM) and PIM Set that finds unserviced nodes and determines the best solution from the kernel. This architecture is scalable and can process graphs with many millions of nodes.

Unless otherwise noted, the words kernel and Processing Element (PE) are used interchangeably. A kernel or PE is the ASIC version of a CPU thread.

Dijkstra's algorithm solves the shortest path problem.

Overview of Dijkstra's Algorithm

1.) Mark all nodes as unvisited. In this implementation, that means shift all nodes into the topmost layer of a tree.

2.) Pick a node to serve as the root. All distances are relative to the root. The root gets a relative distance of 0, while all others are assigned a very large relative distance. In practice, a large number such as 1E30 is typically used to represent infinity.

3.) Calculate the tentative distance of every node neighboring the current node. Add the known distance of the current node to tentative distances so that the relative distance of the current node's neighbors is now known.

4.) Now that the relative distance of all current neighbors is known, remove the current node from the Set.

5.) If there is a destination node, and that destination node has been visited, the algorithm is done.

6.) If, after visiting several nodes, the smallest possible tentative distance is infinity, the algorithm is done.

If Steps 5 or 6 do not terminate the algorithm, take the neighbor with the smallest tentative distance, and make that the new current node.

Some implementations of Dijkstra use a set to store the nodes that have been visited. While that is fine, the lack of order in the data structure should not be taken to mean that nodes can be visited in any order. Because the relative distance is calculated by adding a neighbor's distance to the current node's distance back to the root, nodes must be visited in such a way that. That is why the circuit referred to as a Set in this paper is technically a List with O(1) lookups. That circuit is referred to as a set because its chief functions are to add, remove, and check membership; and the outside world does not need or have access to an address.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

There are some differences worth knowing about between this VHSIC hardware description language (VHDL) prototype and more straightforward C-like implementations.

When we say C-like implementations, we mean single threaded implementations in a C-like language (C++, Java, Python, etc.) on a von Neumann architecture.

C-like implementations sometimes use concurrency to give the illusion (and in some cases, the substance) of parallelism. This VHDL implementation offers true parallelism. The difference matters for proving algorithm correctness. The difference also demands a different memory model than C-like languages have relied on. C-like implementations can sometimes achieve an 8x speedup using 8 cores. But 8 cores would be small by VHDL standards. Because the number of threads in a VHDL can grow far larger than the number of threads in any C-like language (per cm^2), PIM is not merely the next horizon for parallelization; it is a necessity.

With regard to provability, one astute customer asked, “How do I know that this is the best algorithm possible?” The answer is that right now, there is no proof. At the time we were asked to take over this project, there was not enough time in the schedule to do theoretical work and also do the lab work, so we went straight to the lab work. When converting theory to practice, at least one assumption must typically be relaxed.

However, when the assumption that an algorithm is single threaded is not slightly violated but clearly violated, not just 2 or 3 threads but millions, proofs predicated on a Turing machine are untenable. What is needed is a Turing machine with Ethernet – a theoretical computer networked with many other theoretical computers; and to show that that network converges on a correct solution. Although now that we’re talking about an infinite computer network, we also have to address questions such as:

1.) What topology is the network? Random? Small world? Fully connected?
2.) Given that an infinite network can converge on a correct solution, can the network converge on that solution in finite time?
3.) How fast must the size of the network approach infinity compared to the size of the problem?

We have not had time in the schedule to prove that the architecture described here provides the shortest possible path to every node; but if others are interested in tackling these questions, we are happy to collaborate.

A high level overview of the circuitry is illustrated below in Figure 10:

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

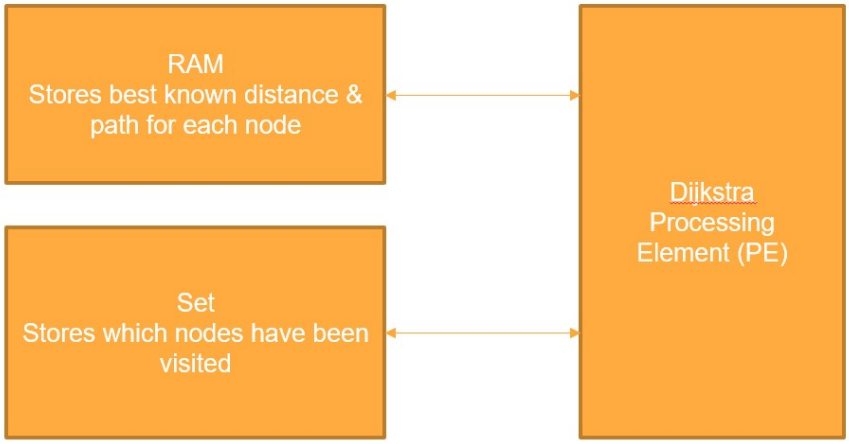

Figure 10: A High Level Drawing of the Dijkstra Algorithm Implemented in VHDL
The RAM and Set have been designed so that many copies of the Processing Element (PE) can run in parallel.

Figure 11 shows the simplest possible connection for a single kernel, or PE. But since this project is about parallelism, Figure 11 illustrates an arbitrary number of PE's all reading from and writing to the same RAM, and reading from the same Set:

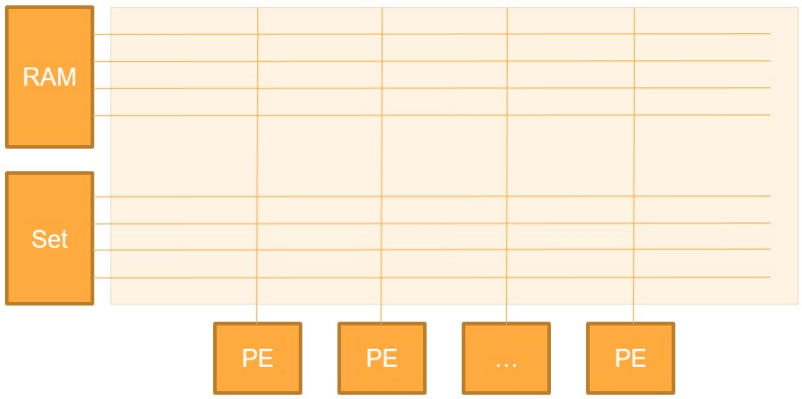

Figure 11: Arbitrary Numbers of PE's can Run in Parallel, Reading from the Same Set, Reading from and Writing to the Same RAM
In practice, the number of PE's may be limited by the density with which venders can pack vias between dies in 3D manufacturing but is not limited by circuit topology

The Dijkstra ASIC Proxy uses 3 kinds of fabric:

1) Conventional ASIC fabric, such as 3-input inverse and gate (NAND) gates,
2) High density RAM with a baked-in merit function to give the outside world the illusion of many write ports; in this case, only service write requests in which the smallest distance is shorter than the best known distance. If several write requests are competing, only service the one with the lowest distance.
3) High density set with many read ports in which each read port is a FIFO. Every time a kernel requests a new node to service, the Set chooses an unserviced node to replace it in

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

> the buffer, then removes that node from the Set. Because order does not matter, and user does not have direct access to any internal memory space, the Set has full ownership over any and all nodes that it is in the process of delegating. This is important because it constrains the scope of the multithreading problem. Instead of notoriously messy "any thread anywhere in the world can modify this database"-like problems, in which meaningful concepts break down (when an infinite number or threads write to the same address at the same time, what is the correct answer, exactly?), the Set designer can realize the interface any way he/she likes.

The author uses the word Set, but technically this data structure is a List with O(1) lookups.

It is referred to as a Set because 1) it serves the same role as a true Set in more conventional implementations of Dijkstra – that of finding an unserviced node and delegating that node to a thread; 2) internally, the operations it uses are add, remove, and check membership; and 3) the user does not need or have access to any address.

One of the keys to making this project work is a careful understanding of the scope: The RAM and the Set do not need to be synthesizable on ASIC fabric. Instead, the RAM and Set need to be realizable in high density field effect transistor (FET) arrays. Since memory vendors have their own tools for synthesizing these kinds of circuits, COTS synthesizers are irrelevant for memory. Therefore, it is important to model a circuit that can be built.

The VHDL Set model uses a RAM internally, but improvements in efficiency are conceivable if engineers can find a circuit that behaves the same way as the VHDL model, without an internal RAM. Such a chip is not commercially available, but can be made.

This circuit also has interesting implications for computer architecture. Now that clock speeds increase slowly (as opposed to the 1990's when they increased exponentially), there is pressure to increase the number of cores in a CPU, yet two challenges stand in the way: 1) thread safety and 2) the von Neumann bottleneck. DARPA and other organizations should consider funding additional research into PIM Sets like the one described here to reduce or solve these challenges. The same goes for heaps, lists, and other data structures. While attempts to implement C-like languages in such a way that the entire program compiles to an FPGA have led to mixed results, mostly failure, (System C works but is quite inefficient) it is reasonable to implement C-like languages in such a way that some design patterns tell the compiler, "Look at ASIC port X at Clock cycle Y, and link that to function Z." In that way, idiomatic code can move subsets of a C-style program to hardware, reducing burden on the CPU without using ASIC resources inefficiently.

The many port RAM has one feature that makes it different than commercially available RAM:

This many port RAM has a baked in merit function that decides which write requests to service.

It is easy to see how such a RAM could greatly accelerate a wide variety of optimization problems. When the burden of deciding which among several alternatives is on a PE, or a CPU,

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

that thread must hog the RAM bus while reading and comparing those alternatives. By placing the merit function in the memory, the RAM bus is freed up so that other threads can use it.

Interestingly, the baked-in merit function allows every core to have its own RAM port, which reduces the likelihood that cores have to share a RAM bus in the first place. When combined with 3D manufacturing (stacking a RAM die on top of an ASIC die), the von Neumann bottleneck may be eliminated.

It is also interesting that the PIM RAM doesn't need to know or care how the merit score was calculated. All it does is say, "If your score is greater than the score already in memory, I will service the write request. Otherwise, I throw it out." The PIM RAM is agnostic to the merit function itself. It simply uses the > operator to decide which requests are worth servicing. When multiple competing requests have the exact same score, the PIM RAM can choose the winner by lottery or some secondary criterion. After all, if several write requests have the exact same score, it is fair to assume that they are about equally good, even if they are different values. Therefore, whatever circuit is feeding the PIM RAM can decide how to assign the score any way it likes. This feature makes PIM RAMs with baked in > operator program agnostic. They can store data for any program that can accept their cost-benefit profile.

By 1960, RAMs were the dominant memory circuit, presumably because a RAM can emulate any other data structure, but not the other way around. Some other data structures are commonly implemented in circuitry. For example, almost all CPU's have a hardware stack. Punch cards are probably the most popular form of hardware list (obviously not used much anymore); ring buffers easily implement a list in VHDL, although that is not common in practice.

Sets in VHDL are easy in simulation but challenging for FPGA fabric. VHDL can model a set as shown in Figure 12:

```
for i in 0 to NUM_NODES-1 loop
    if i /= ROOT then
        if v_num_rows_read >= NUM_PORTS then
            exit;
        else
            if s_set_ram(i) = seek_state then
                s_output_queue(v_num_rows_read).node_num <= i;
                s_output_queue(v_num_rows_read).is_valid <= '1';
                v_num_rows_read := v_num_rows_read + 1;
            end if;
        end if;
    end if;
end loop;

for i in 0 to NUM_PORTS-1 loop
    if i_read_requests(i).read_enable = '1' then
        s_membership(i).node_num <= i_read_requests(i).node_num;
        s_membership(i).is_member <= s_set_ram(i_read_requests(i).node_num);
        s_membership(i).is_valid <= '1';
    else
        s_membership(i) <= BLANK_SET_OUTPUT_PORT;
    end if;
end loop;
```

Figure 12: A Naïve C-like Model of a Set in VHDL

This model is C-like because it uses for loops in ways that lead to exploding circuit size in VHDL.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Synthesizing the code in Figure 12 would not go well. A better approach to synthesizing such a circuit would be a tree, as illustrated in Figure 13, below:

In Figure 13, only the topmost layer stores hashes. All lower layers are solely to aggregate information. Each read port requires its own aggregation tree. This circuit calculates membership in logN(M) clock cycles; but if N >> 2, logN(M) looks increasingly like O(1). For instance, N=128 would yield a circuit that, for all practical purposes, is O(1). For example, if the top layer already contains 134M nodes (~= 2^27), and we give the tree an extra clock cycle, the top layer would have to increase 128-fold, to about 17B nodes to use that extra clock cycle. The 134M node circuit requires 4 clock cycles, as shown in equation below:

Ceil(log(134000000) / log(128)) = ceil(3.85) = 4)

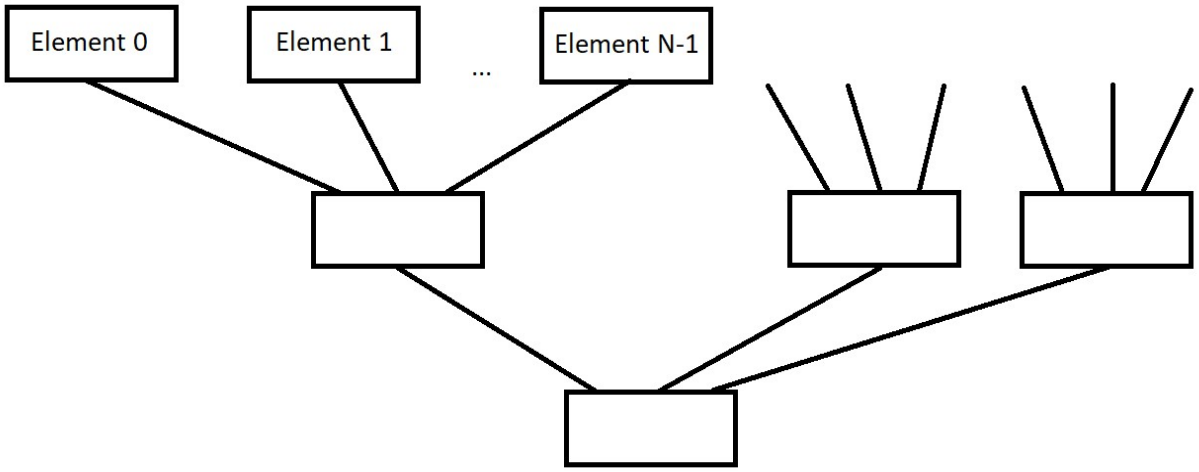

Figure 13: A Tree Implementing a Set in VHDL

If multiple ports try to add and remove the same element at the same time, technically this circuit would require a merit function to resolve such conflicts. Such a function is unnecessary for Dijkstra, for two reasons: 1) Nodes are added before running the algorithm, and while running the algorithm, nodes are only removed; and 2) it is trivial to make the top layer a shift register. Therefore, the user can shift many hashes into the top layer in O(N) time; then check membership effectively in O(1) time.

It is interesting to note that if the user is willing to increase hardware arbitrarily, the runtime floor for this circuit becomes shifting the data into the topmost layer of the tree. But the shifting effort can be parallelized, which drives the coefficient of O(N) down arbitrarily. Von Neumann architectures do not have this property – not even with huge numbers of CPUs. The reason is that thread safety eventually becomes very hard. As long as the burden of finding unvisited nodes is on CPUs, each CPU has to clog its RAM bus in order to find that unvisited node. Therefore, no amount of CPUs can implement Dijkstra as efficiently as the architecture described here. That is

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

why DARPA and other organizations should consider funding additional research into high density hardware sets.

The key to parallelizing Dijkstra-like algorithms is eliminating the von Neumann bottleneck.

In this paper, we have achieved that by modeling a PIM RAM and PIM Set that take the work of finding unserviced nodes and determining the best solution from the kernel. We have also designed a kernel that can be synthesized on FPGA or ASIC fabric, and many instances of the kernel can talk to the PIM RAM and Set. This architecture is scalable and can process graphs with many millions of nodes.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

4 Auction Algorithm

The auction algorithm solves the assignment problem. Many instances of the kernel described in this report can be parallelized in a scalable architecture which increase throughput beyond what CPUs can achieve. The finite state machine (FSM) described coordinates the addresses, write enables, and other low-level details that control the RAMs, merit function, and best bid module. The PPE is lightweight, scalable, and can solve large assignment problems. The best bid module makes a convenient, scalable, independently verifiable module that tracks the best known actor against an arbitrary merit function.

Auction algorithms solve Computer Science & Engineering problems analogous to a real life auction. In an auction, there are buyers and sellers, users and assets. Auction algorithms match each buyer with a resource according to how useful a resource is to a buyer. Auction algorithms are useful for optimizing a wide variety of problems that involve matching up some user with some resource. For example, passengers with taxis, servers with clients, or soldiers with satellites. Auctions provide a solution to complex, multi-variate optimization problems. Auction algorithms are most useful in domains which have:

1.) Several inputs – OR –
2.) A large number of actors, resources, or both, – AND –
3.) No obvious solution.

There are two fundamental approaches for implementing the auction algorithm:

1.) Jacobi Architecture: 1 actor bids on every resource
2.) Gauss-Seidel (GS) Architecture: Every actor bids on 1 resource.

For reasons described in [2], GS makes for a more efficient FPGA implementation.

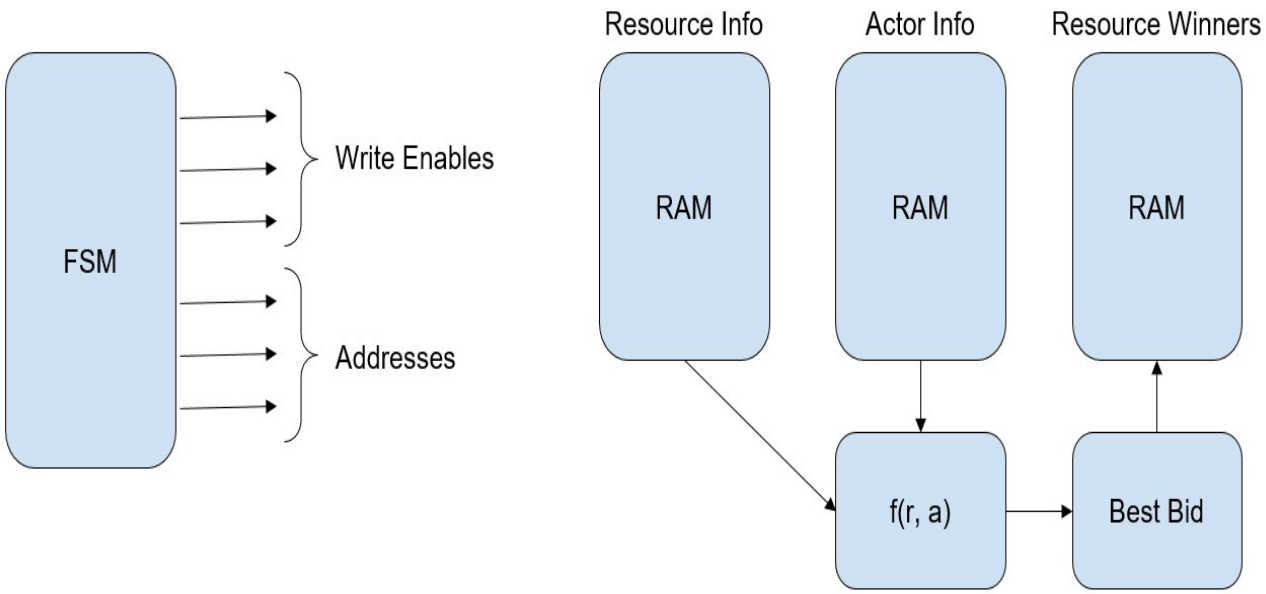

Figure 14: The Gauss-Seidel Architecture Described in [1]

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

At their core, auction algorithms are simple. They require:

1) A merit function which decides how valuable an asset is to a particular buyer, and
2) A list of buyers,
3) A list of sellers.

Any auction algorithm outputs a list of tuples, such that for each tuple,

(Actor, Resource)

in equation 2, the resource is the most valuable among available resources to the actor – with the constraint that actors who have already won an asset do not bid again. This is an important difference between auction algorithms and real life ones, where a single organization can buy all of a given resource. In a computer auction, we typically do not one server to snatch up all the clients or vice versa.

As a simple example of an auction algorithm on a von Neumann architecture, consider the following:

```
// Cpp-like pseudocode for auction algorithm.
// buyersThatWon is a set
// resourcesWhichWereWon is a set
// myHashMap is a hash map.
// valueOfThisResource is typically long or double.
// meritFunction determines the value of a given resource to a given buyer.

for (int buyerNum = 0; buyerNum < someSize; ++buyerNum) {
    greatestKnownValue = NEGATIVE_INFINITY;
    for (int resourceNum = 0; resourceNum < someSize; ++resourceNum) {
        // Check if this buyer has won anything. If not, proceed:
        if (!buyersThatWon.contains(buyerNum)) {
            // Check if this resource has been won. If not, proceed:
            if (!resourcesWhichWereWon.contains(resourceNum)) {

                valueOfThisResource = meritFunction(buyerNum, resourceNum);
                if (valueOfThisResource > greatestKnownValue(buyerNum)) {
                    greatestKnownValue = valueOfThisResource;
                    bestResourceNum = resourceNum;
                }
            }
        }
    }
    buyersThatWon.add(buyerNum);
    resourcesWhichWereWon.add(bestResourceNum);
    myHashMap.set(buyerNum, bestResourceNum)
}
```

Figure 15: C-like Pseudocode for an Auction Algorithm

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

This example runs in O(N^2) time; uses two sets with O(1) read and write times; and one hash map also with O(1) read and write times. Therefore, the overall time complexity of this example is O(N^2). There are things we could do to make our C implementation more efficient, but that would be chasing the wrong goal. The correct goal is to make a VHDL implementation that allows the user to throw fabric at the problem until it runs faster than a CPU can for the same electric bill, real estate, or mass.

The I/O ports for an auction kernel are shown below in Figure 16:

In order to coordinate the timing of RAM reads and writes, and execution of merit function illustrated in Figure 16, a FSM was developed. The state table is shown below in Tables 2 and 3. The Moore outputs from the FSM are shown in Tables 4 and 5. For the sake of brevity, Moore outputs are not shown for Fast/Full RAM erasure states.

```
entity ProcessingElement is
    generic(
        error_if_warning_ignored : std_logic := '0'
    );
    port(
        clk : in std_logic;
        rst : in std_logic;
        i_soft_rst : in std_logic := '0';
        i_full_ram_clear_when_contest_done : in std_logic := '0';
        i_clk_en : in std_logic := '1';
        i_direct_write_mode : in std_logic; -- Use this to write actors & resources directly to RAM
        i_resource_din : in std_logic_vector(actor_and_resource_data_width-1 downto 0);
        i_actor_din : in std_logic_vector(actor_and_resource_data_width-1 downto 0);
        i_addr_din : in std_logic_vector(addr_width-1 downto 0);
        i_start_contest : in std_logic;
        i_start_readout : in std_logic;
        o_reading_out : out std_logic;
        o_resource_output : out std_logic_vector(actor_and_resource_data_width-1 downto 0);
        o_winner_output : out std_logic_vector(actor_and_resource_data_width-1 downto 0);
        o_readout_done : out std_logic;
        o_num_resources_written : out unsigned(addr_width-1 downto 0);
        o_winner_addr_output : out unsigned(addr_width-1 downto 0);
        o_readout_addr : out unsigned(addr_width-1 downto 0);

        -- Warnings & Errors:
        o_ignored_direct_write_warning : out std_logic;
        o_ignored_direct_write_warning_sticky : out std_logic;
        o_fsm_error_sticky : out std_logic;
        o_fsm_error : out std_logic;
        o_thinking : out std_logic;
        o_running_contest : out std_logic;
        o_contest_done_start_readout : out std_logic;
        o_self_reset : out std_logic;
        o_actors_used_addr : out std_logic_vector(addr_width-1 downto 0); -- For testbench. Synthesis not recommended.
        o_actors_used_we : out std_logic -- For testbench. Synthesis not recommended.
    );
end ProcessingElement;
```

Figure 16: The Interface for an Auction Kernel

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Table 2. State Transition Table for the Auction FSM

Reset		Loiter
Loiter		Waiting for Data
Waiting for Data	i_direct_mode_write = '1'	Receiving Data
Receiving Data	i_direct_mode_write = '0'	Waiting to Start
Waiting to Start	i_start_contest = '1'	Starting Contest 1
Starting Contest 1		Starting Contest 2
Starting Contest 2		Starting Contest 3
Starting Contest 3		Starting Contest 4
Starting Contest 4		Starting Contest 5
Starting Contest 5		Running Contest
Running Contest	S_inner_idx >= s_inner_idx_rollover – 3	Pausing Contest 1
Pausing Contest 1		Pausing Contest 2
Pausing Contest 2		Pausing Contest 3
Pausing Contest 3		Pausing Contest 4
Pausing Contest 4		Pausing Contest 5
Pausing Contest 5		Write Winner 1
Write Winner 1		Write Winner 2
Write Winner 2		Write Winner 3
Write Winner 3		Write Winner 4
Write Winner 4		Write Winner 5
Write Winner 5		Reset Best Bid Info 1
Reset Best Bid Info 1		Reset Best Bid Info 2
Reset Best Bid Info 2		Reset Best Bid Info 3
Reset Best Bid Info 3	(s_outer_idx >= s_outer_idx_rollover – 1)	Waiting for Readout
Reset Best Bid Info 3	Else	Reset Best Bid Info 4
Reset Best Bid Info 4		Reset Best Bid Info 5
Reset Best Bid Info 5		Unpausing Contest 1

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Table 3. State Transition Table for the Auction FSM

Present State	Condition	Next State
Unpausing Contest 1		Unpausing Contest 2
Unpausing Contest 2		Unpausing Contest 3
Unpausing Contest 3		Unpausing Contest 4
Unpausing Contest 4		Unpausing Contest 5
Unpausing Contest 5		Running Contest
Waiting for Readout	i_start_readout = '1'	Reading Out
Reading Out	s_readout_addr >= s_outer_idx_rollover - 1	Readout Done
Readout Done	s_prefer_full_ram_erase = '1'	Full RAM Erase 200
Readout Done	i_direct_write_mode = '1'	Fast RAM Erase 100
Fast RAM Erase 100		Fast Ram Erase 101
Fast RAM Erase 101	(i_direct_write_mode = '0') && (s_ram_erasure_addr < s_prev_inner_idx_rollover)	Fast RAM Erase 102
Fast RAM Erase 101	(i_direct_write_mode = '0') && (s_ram_erasure_addr < s_prevOuterIdxRollover)	Fast RAM Erase 102
Fast RAM Erase 101	(i_direct_write_mode = '0') && (i_start_contest = '1')	Starting Contest 1
Fast RAM Erase 101	(i_direct_write_mode = '0') && (None of the above)	Waiting To Start
Fast RAM Erase 102	(s_ram_erasure_addr >= s_prevOuterIdxRollover) && (s_ram_erasure_addr >= s_prevInnerIdxRollover)	Fast RAM Erase 103
Fast RAM Erase 103		Waiting to Start
Full RAM Erase 200		Full RAM Erase 201
Full RAM Erase 201	s_ram_erasure_addr = (All 1's)	Waiting for Data

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Table 4. Moore Output Table for the Auction FSM

Present State	Output Assignments
Reset	Everything set to 0
Loiter	Everything set to 0
Waiting for Data	s_okay_to_direct_write <= 1 Everything else set to 0
Receiving Data	s_data_ack <= 1 s_num_actors_received += 1 s_num_resources_received += 1 s_outerIdxRollover += 1 s_innerIdxRollover += 1 Everything else set to 0
Waiting to Start	Everything set to 0 that was not incremented in Receiving Data
Starting Contest 1	s_outerIdx <= "000…01" s_innerIdx <= "000…01" s_enable_valuation <= '0'
Starting Contest 2	s_enable_valuation <= '1'
Starting Contest 3	s_outerIdx <= "000…01" s_innerIdx <= "000…01" s_enable_valuation <= '1'
Starting Contest 4	s_outerIdx <= "000…01" s_innerIdx <= "000…01" s_enable_valuation <= '1'
Starting Contest 5	s_outerIdx <= "000…01" s_innerIdx <= "000…01" s_enable_valuation <= '1'
Running Contest	s_pricing_resource_number <= s_outerIdx s_innerIdx += 1 s_contest_in_play <= '1' s_thinking <= '1' s_enable_valuation <= '1'
Pausing Contest 1	s_innerIdx += 1
Pausing Contest 2	
Pausing Contest 3	
Pausing Contest 4	
Pausing Contest 5	
Write Winner 1	
Write Winner 2	
Write Winner 3	
Write Winner 4	
Write Winner 5	

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

Table 5. Moore Output Table for the Auction FSM

Reset Best Bid Info 1	s_innerIdx <= “000…01” s_enable_valuation <= ‘0’
Reset Best Bid Info 2	
Reset Best Bid Info 3	
Reset Best Bid Info 4	
Reset Best Bid Info 5	
Unpausing Contest 1	s_outerIdx += 1
Unpausing Contest 2	
Unpausing Contest 3	
Unpausing Contest 4	
Unpausing Contest 5	
Waiting for Readout	s_prevOuterIdxRollover <= s_outerIdxRollover s_prevInnerIdxRollover <= s_innerIdxRollover s_contest_done_start_readout <= ‘1’ Everything else set to 0

Reading Out	s_reading_out <= ‘1’ s_thinking <= ‘1’ s_contest_in_play <= ‘0’
	s_readout_addr += 1
	
	if (s_readout_addr >= s_outerIdxRollover) {
	s_readout_done <= ‘1’
	} else {
	  s_readout_done <= ‘0’
	}

Readout Done	s_readout_done <= ‘1’ Everything else set to 0

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

In order to track the best known bid on a particular resource, the Kernel uses a Best Bid module. Its I/O ports are shown below in Figure 17:

```
entity BestBid is
    port (
        clk : in std_logic;
        rst : in std_logic;
        en : in std_logic;
        i_soft_rst : in std_logic;
        i_actors_used_dout : in std_logic_vector(0 downto 0);
        i_one_bid : in unsigned(price_width-1 downto 0);
        i_one_actor : in std_logic_vector(actor_and_resource_data_width-1 downto 0);
        i_actor_address : in std_logic_vector(addr_width-1 downto 0);
        o_best_bid : out unsigned(price_width-1 downto 0);
        o_prospective_winner : out std_logic_vector(actor_and_resource_data_width-1 downto 0);
        o_best_address : out std_logic_vector(addr_width-1 downto 0);
        o_winner_ram_din : out std_logic_vector(actor_and_resource_data_width-1 downto 0)
    );
end BestBid;
```

Figure 17: I/O Pins for the Best Bid Module

The job of the best bid module is simple:

- When i_soft_rst = '1', set the best bid to all 1's. (Best bid is unsigned.) This implementation makes lower scores superior. This convention is more convenient because the merit function is a simple geographic distance between 2 points on a Cartesian coordinate system. A vendor could just as well make higher scores superior.
- When a new bid is discovered that is superior compared to the previous best known bid, the new best bid is recorded, along with the actor who made that bid. This new actor is recorded to as the prospective winner. The address of the prospective winner is also recorded as the best address.
- When the contest is over, the prospective winner becomes the winner.
- There is a separate contest for each resource. The FSM shown above in Figure 17 strobes the i_soft_rst pin on the Best Bid module to start a new contest for the next resource.

This implementation of the auction algorithm solves the assignment problem. Many instances of the kernel can be parallelized in order to increase throughput. In this simulation, conventional RAMs were used. The FSM coordinates the addresses, write enables, and other low-level details that control the RAMs, merit function, and best bid module. The best bid module makes a convenient, scalable, independently verifiable module that tracks the best known actor against an arbitrary merit function.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

5 RESULTS

SDH ASIC proxy results are shown below in Table 6. The core power was estimated using the Quartus power analyzer tool. The ASIC performance was estimated by using guidelines provided by Intel for structured ASICs. The structured ASIC approach allowed for conservative estimates that seemed reasonable for performance estimates. Structured ASICs typically have a 20% increase in Fmax and anywhere from a 50%-80% reduction in power. Using these numbers, the ASIC estimates in the right most columned were derived.

Table 6. ASIC Proxy Results

Kernel	Device	Fmax (FPGA)	Core Power (FPGA)	FPGA Efficiency	ASIC Estimates*
FFT	Stratix 10	750 MHz	10.2 Watts	347 GFLOPs/W	700 GFLOPs/W
2D Convolution	Stratix 10 MX	750 MHz	9.6 Watts	540 GFLOPs/W	1.1 TFLOPs/W
Dense MM	Stratix 10	600 MHz	25 Watts	610 GFLOPs/W	1.2 TFLOPs/W
Sparse MM	Stratix 10	182 MHz	15 Watts	2.5 GFLOPs/W	5 GFLOPS/W
Dijkstra	Kintex 7	100 MHz	12 Watts	430 MFLOPs/W	860 MFLOPs/W
Auction	Kintex 7	100 MHz	9.5 Watts	290 MFLOPs/W	580 MFLOPs/W

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

6 FINANCIALS

Figure 18 shows the SDH ASIC Proxy research financials.

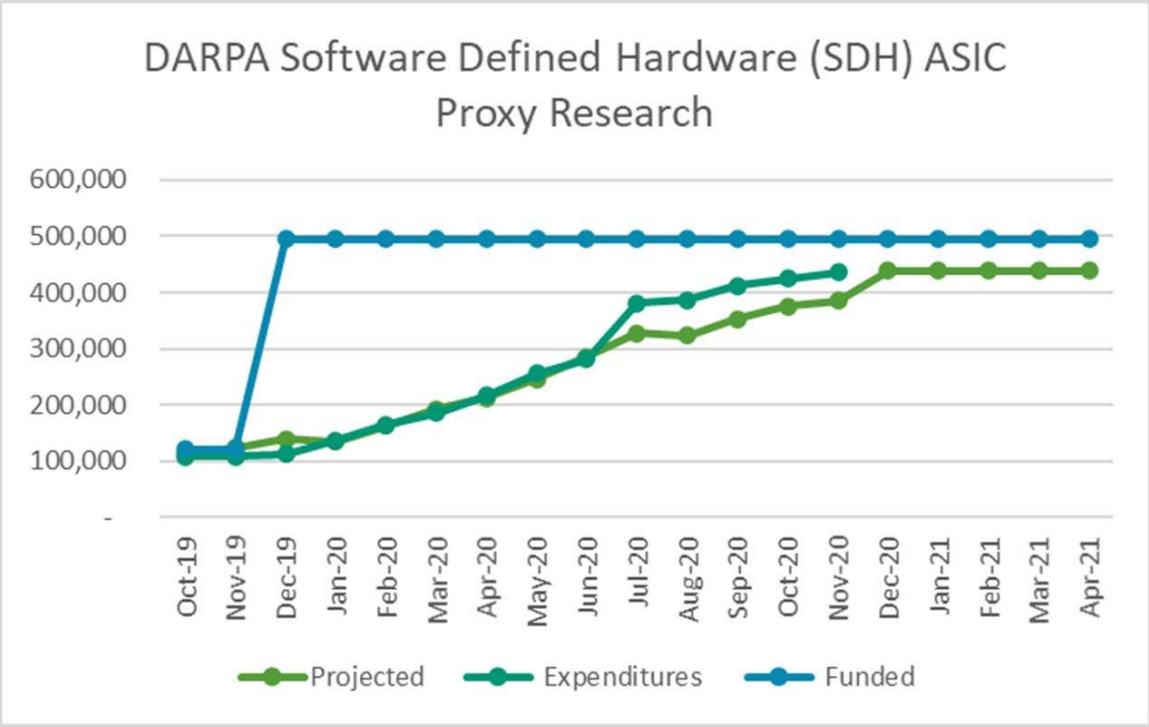

Figure 18: SDH ASIC Proxy Research Financials

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

7 REFERENCES

1. J.-W. a. S. B. C. Jang, "Energy and Time-Efficient Matrix Multiplication on FPGAs," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2005.

2. Z. e. a. Zhu, "An FPGA-Based Acceleration Platform for Auction Algorithm," Vols. 978-1-4673-0219 IEEE.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

ACRONYM	DESCRIPTION
ASIC	Application Specific Integrated Circuit
CPU	Central Processing Unit
CSR	Compressed Sparse Row
DARPA	Defense Advanced Research Projects Agency
DSP	Digital Signal Processing
FET	Field Effect Transistor
FFT	Fast Fourier Transform
FIFO	First in First Out
Fmax	Maximum Frequency
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GB	Gigabyte
GFLOPS	Giga Floating Point Operations
GPU	Graphics Processing Unit
GS	Gauss-Seidel
IO	Input Output
NAND	Inverse And Gate
NxN	Array with N rows and N columns (N is variable)
PE	Processing Element
PIM	Processor in Memory
RAM	Random-Access Memory
SDH	Software Defined Hardware
TBD	To Be Determined
UDRI	University of Dayton Research Institute
VHDL	VHSIC Hardware Description Language

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.