CPRE 563 Project Proposal

Jake Hafele, William Zogg
Department of Electrical and Computer Engineering
Iowa State University, Ames, IA 50014

Abstract

Many previous works in the past have highlighted the benefits of FPGA acceleration for high bandwidth applications, such as machine learning. But, while hardware specific accelerators in FPGA and ASIC applications can speed up the computation time and bandwidth for a specific task, the latency times can still be large. With the rise in Computational Storage Devices (CSDs), including SmartSSD, a new solution and platform can be utilized, which closely integrates a SSD storage device and FPGA on the same system. By utilizing a neural network image inference application, we can determine the benefits of SmartSSD for high bandwidth applications, which may typically be accelerated on a traditional FPGA accelerator platform.

I. Introduction and Motivation

With the rise of Machine Learning (ML) in computers, increasing the throughput of these systems is crucial in enabling their use in real situations [1]. One such use is image classification, which is not just computationally taxing, but memory-usage too. FPGAs, with their long-standing presence, have helped accelerate the computing tasks related to image classification, but have not helped the bandwidth (BW) demands of ML inferences [2].

Recent years have seen the development of Computational Storage Devices (CSD) [3]. With their unique combination of speedy flash and a local FPGA, the SmartSSD offers a way to accelerate ML inferences by computing them close to their storage location, without dealing with the memory pipeline latency in a normal CPU-based system [4]–[6].

In this project, we aim to explore the benefits of a SmartSSD implementation of a Neural Network (NN) accelerator vs a traditional CPU system with a local SmartSSD from ISU. Using a baseline NN for classifying images via TinyImageNET, we will first evaluate the throughput and accuracy of the host PC. Then, we will evaluate the performance benefits seen from running the inference on the SmartSSD by reducing the BW bottleneck we would normally expect from our host PC.

II. Related Works

A. Neural Network Inference with SmartSSD

There are multiple related works surrounding the application of Neural Network Inference, similar to our own Neural Network library, that have already been created. These can act as a baseline for how to model our code within the SmartSSD architecture.

In one such work, an architecture is proposed which loads the weights of a neural network layer directly from the SSD device within SmartSSD, and the other inputs are loaded over a Peer to Peer buffer from the host to the FPGA [7]. The purpose of this architecture is to perform accelerated neural network recommendation operations. Computations are then performed within the FPGA, and then written back to the SSD device over the shared PCIe bus.

Another related work involves applying the kNN algorithm on the FPGA within SmartSSD [8]. This work utilizes Xilinx OpenCL, which is the same target methodology we are using for our own project.

B. Other SmartSSD Applications

NeSSA provides another machine learning sample, with a focus on machine learning subset training, to achieve a speedup in the process by reducing the dataflow [5]. This work also achieves a higher performance by utilizing SmartSSD to achieve lower latency between the storage device and computation, by utilizing near-storage computation with SmartSSD.

Many other applications have also been utilized by SmartSSD, such as data sorting [9]–[11] and query processing applications [12], [13].

III. Solution Approach

Our goal will be solved by utilizing an existing neural network library that was developed in another class at Iowa State, CPRE 487/587X, and applying it to the SmartSSD platform. We will measure and compare results for running an image classification on a host platform and with SmartSSD to compare the speedup for multiple image sets. We are expecting to see marginal improvement with the SmartSSD interface due to higher bandwidth allowance between the SmartSSD and FPGA, alongside being able to utilize more memory with the FPGA, unlike with the normal amount of DRAM on an FPGA SoC.

A. Neural Network with Host

As a baseline for measuring performance gains in SmartSSD, we will be running a neural network implementation from CprE 487/587X on the host SmartSSD server. This neural network combines several convolution, dense, pooling, and soft-max layers to classify images in the TinyImageNET dataset. We will measure performance using the throughput of the network. Once this neural network is up and running on the host PC, we will utilize Vitis to run the model on the SmartSSD without FPGA acceleration, making necessary adjustments in the code compilation for it to run properly.

B. Convolution Kernel

Once the original library is confirmed to work on the SSD apart of SmartSSD on the host server, we can begin working on synthesizing the C code for our kernel design. This will require us to refactor our design for the convolution layer with the longest execution time, Layer 2, so that a new class is made for this specific convolution layer. We can then break down the convolution process such that a small enough kernel can be synthesized and loaded onto the SmartSSD FPGA with our target design. We will also gain timing and utilization reports to estimate if our target kernel design is feasible in the Kintex SmartSSD FPGA.

C. SmartSSD Application

Our first steps with the SmartSSD platform will be to learn how to interface with the product. Another team working with SmartSSD has added us to a Discord server and created a user account on the SmartSSD server. We have successfully gained access to the server and gotten Vivado and Vitis to run on the server through a server login. With this, we can run validate commands to verify communication with the SmartSSD utilizing xbutil commands.

Next, online resources and sample repositories will be referenced for examples of how to use Peer to Peer connection with the host code for our design. This will enable us to communicate between the Host, SmartSSD, and FPGA through the PCIe switch inside of SmartSSD. There are many sample projects which can be both simulated and ran on the SmartSSD with our target synthesizable convolution kernel.

D. SmartSSD Analysis

There are multiple existing works relating to SmartSSD, which we can use as a basis to surround our presentation discussion and results analysis. Our main performance evaluation will focus on the execution time speedup for the neural network inference between the host and SmartSSD platform, as seen in other works [4]. Other works analyze the training time for large machine learning datasets, in which propogation delay is compared against computation time. [5]. We believe this would be an interesting opportunity to compare the latency delays between the host and SmartSSD data transfer, alongside the additional bandwidth with SmartSSD.

E. Summary

Our work has the following key contributions:

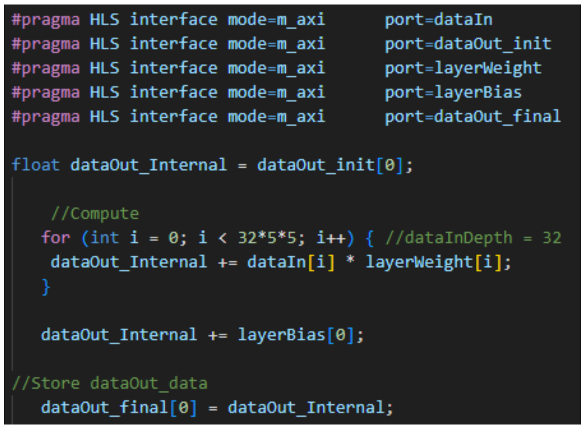
- Verify existing CPRE487/587X neural network interface works on SmartSSD server host
- Refactor library around synthesizable convolution kernel
- Develop synthesized convolution kernel which can be flashed on a Kintex FPGA
- Create reproducible SmartSSD host framework to write to the SmartSSD, and perform direct peer to peer operations between the SSD and FPGA within SmartSSD
- Analyze execution time and latency delay between the baseline Host CPU and SmartSSD kernel application

IV. Methodology

A. Neural Network with Host

To run the CPRE487 library on the Host server on the SmartSSD, we had to repartition and mount

SmartSSD. By calling *lsblk -lf* we were able to identify the SmartSSD SSD device name, which was *nvme0n1*. We utilized *fdisk* to delete the existing old partitions and create a new one with 20 Gb of size for our group. This size provided enough space to install our full codebase and store binary files that were used in the layer operations. After formatting the new partition with *mkfs*, we were able to mount the partition to the Linux server with *mount* to our desired directory. Within this mounted partition, we installed our codebase and run our baseline unaltered software to obtain timing results of the original design on the SmartSSD storage device, without any FPGA kernel running.



```
#pragma HLS interface mode=m_axi      port=dataIn
#pragma HLS interface mode=m_axi      port=dataOut_init
#pragma HLS interface mode=m_axi      port=layerWeight
#pragma HLS interface mode=m_axi      port=layerBias
#pragma HLS interface mode=m_axi      port=dataOut_final

float dataOut_Internal = dataOut_init[0];

    //Compute
   for (int i = 0; i < 32*5*5; i++) { //dataInDepth = 32
    dataOut_Internal += dataIn[i] * layerWeight[i];
   }

   dataOut_Internal += layerBias[0];

//Store dataOut_data
   dataOut_final[0] = dataOut_Internal;
```

Figure 1: Original Kernel Code

B. Convolution Kernel

Before we could begin synthesizing any code for our Convolution kernel, we first had to refactor the existing code for the target convolution layer, layer 2, such that we could strip it down to the most basic function. To do this, we created a new class for the Layer 2 convolution, titled Convolution HLS (High Level Synthesis), with the goal of synthesizing the lowest level of the operation. In the original design, the three innermost nested for loops in the design contain a convolution operation over a single output data point. We determined this was the best function to use for our kernel since it would provide 800 pipelined computations, which was small enough to fit on the FPGA, while still providing room for benefit with high bandwidth reads to/from the FPGA/SSD.

For this, a new function *computePointHLS* was designed, as seen in Figure 1. Designing this function was challenging, since we were required to flatten the existing function arguments of 3D and 4D floating point arrays to 1D arrays, to ensure that the code could be synthesized in Vitis HLS and use existing memory interfaces in Vitis. This made the computation much easier to synthesize, but required more refactoring in the parent Convolution function, which used a data struct with multidimensional 4D arrays. To flatten this, we utilized internal 1D arrays which would be rebuilt for every convolution point, to act as inputs to computePointHLS.

With the code designed and rerun, we could verify the same functionality after refactoring, to guarantee we did not lose any accuracy due to coding errors. Multiple challenges came up when engaging with Vitis HLS, to synthesize the target kernel *computePointHLS*. One such being the communication interfaces. After researching different Vitis HLS memory interfaces, we determined *m_axi* would be appropriate for streaming a series of array values as a pointer input.

After running Vitis HLS synthesis, we recieved a synthesis report which gave us estimated timing performance for the kernel, and utilization of the FPGA kernel which would be flashed to the SmartSSD Kintex FPGA.

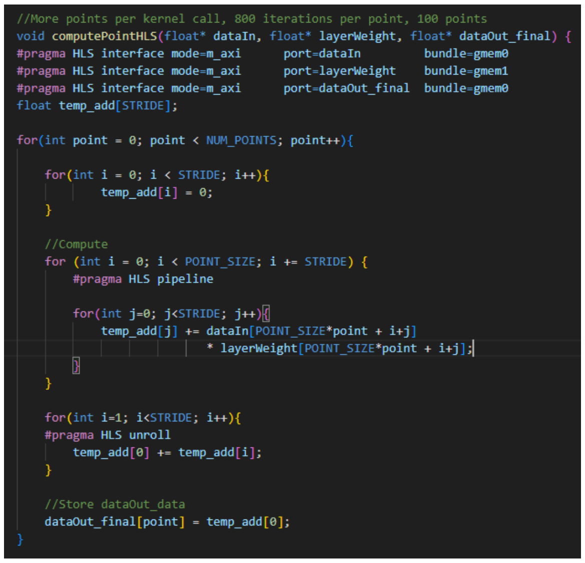
After running the first kernel synthesis, we learned from the reports that the computation time was far greater than we expected. This was due to unoptimal floating point calculations which had greater than 1 cycle of delay, which led to issues when trying to pipeline the target design. We also faced challenges with under utilizing the data bandwidth and overall memory storage of the system, and we ended up running more computation points per kernel call. Specifically, we determined that we could fit up to 1024 kernel calls worth of data into the FPGA at once in the local DRAM inside the FPGA. We measured kernel calls for 40, 90, and 1024 points in our final results. This meant that we needed to store 1,638,400 floating point input values in our FPGA at one time, versus the initial 1,600. This led to much less overhead in timing.

The following optimizations were made on the kernel:

- Using 2 seperate m_axi input busses
- Utilizing Loop Unrolling to parallelize computation of single convolution point

- Executing multiple convolution points in one kernel call

The code in Figure 2 represents the optimized kernel, which incorporates loop unrolling (32 values in parallel), input axi memory busses (2), and another for loop to run for NUM_POINTS (1024) convolution point iterations.



```
//More points per kernel call, 800 iterations per point, 100 points
void computePointHLS(float* dataIn, float* layerWeight, float* dataOut_final) {
#pragma HLS interface mode=m_axi      port=dataIn         bundle=gmem0
#pragma HLS interface mode=m_axi      port=layerWeight    bundle=gmem1
#pragma HLS interface mode=m_axi      port=dataOut_final  bundle=gmem0
float temp_add[STRIDE];

for(int point = 0; point < NUM_POINTS; point++){

    for(int i = 0; i < STRIDE; i++){
            temp_add[i] = 0;
    }

    //Compute
    for (int i = 0; i < POINT_SIZE; i += STRIDE) {
        #pragma HLS pipeline

        for(int j=0; j<STRIDE; j++){
            temp_add[j] += dataIn[POINT_SIZE*point + i+j]
                            * layerWeight[POINT_SIZE*point + i+j];
        }
    }

    for(int i=1; i<STRIDE; i++){
    #pragma HLS unroll
        temp_add[0] += temp_add[i];
    }

    //Store dataOut_data
    dataOut_final[point] = temp_add[0];
}
```

Figure 2: Optimized Kernel Code

C. *SmartSSD Application*

We began interfacing with the SmartSSD by calling *xbutil validate*, which would verify read/write functionality to SmartSSD, to ensure that the device was programmed and connected properly to the Host server.

Next, we developed a new Application Project in Vitis, with a sample repository provided called *P2P Simple*. This design included a simple adder kernel which would write values from the Host to the FPGA, compute the kernel, and read back from the FPGA. We modified this design, with the help of Anthony Manschula, to include an interface between the SSD and FPGA directly, with the intent of measuring latency times between the two. We were able to modify the host code under *Host.cpp*, and insert the target synthesizable kernel code from before, *computePointHLS*. Vitis provides a 1 stop shop which can synthesize and build the kernel, host code, and interface to flash the Kintex FPGA on SmartSSD.

By utilizing the existing OpenCL library, we were able to set up Peer To Peer (P2P) Buffers between the Host/SSD and SSD/FPGA. This allowed us to read/write to the SSD to load in test data, but also directly interface between the SSD/FPGA without the interference of the Host computer. This distinction allows us to discreetly measure timing results of each stage, as described below.

In the host code, the process runs as follows:

1) Initialize Kernel, SmartSSD, Sim information
2) Write data from Host to SSD in fixed blocks
3) Read input arguments from SSD to FPGA
4) Activate kernel in FPGA
5) Write kernel output from FPGA to SSD
6) Read value from SSD to Host to Verify computation

D. *Measuring Results*

The target of the SmartSSD design is to measure the improvement in data movement delay, so our design utilizes the *chrono* C++ library to measure the time it takes to read data into the FPGA, compute the kernel, and write data back into the SSD. By measuring each of these steps individually, we can determine the speedup for each process, as well as an overall execution time compared to the baseline CPU implementation. Additionally, we modified the original baseline code to measure the same latencies for reading, computing, and writing data back for the floating point operations.

V. Results

A. *Neural Network with Host*

Table I demonstrates the results of running the CPRE487 Library on the Host server mounted to the SSD device. Since Layer 2 has the longest latency, and takes up 68.28% of the total execution time, we determined our target layer to derive a kernel should come from Convolution Layer 2 in our library.

B. *Convolution Kernel*

After synthesizing our design for the SmartSSD application, a synthesis report for the kernel is generated within Vitis. The original kernel design computed a single convolution point of 800 values in 9107 cycles, with an estimated latency of 30.354 microseconds. The full convolution layer computes 100,352 points, which would lead to an overall delay of around 3.04 seconds

Layer	Time(ms)
1	47.834
2 (Target)	662.260
3	72.784
4	130.879
5	0.210
6	22.695
7	31.256
8	0.045
9	0.011
10	1.646
11	0.220
12	0.0214
Total	969.861

Table I: Host Code Layer Latency

on its own, excluding the overhead of the kernel calls themselves from the host code. Since the entire application was aiming to run faster than near 0.66 seconds, this was unacceptable, and a main motivator in improving the kernel with more computations, higher memory utilization, and wider memory busses.

The final optimized kernel, referenced in Figure 2, took 708,747 cycles to complete, with an estimated execution time of 2.362 milliseconds. While this is a marginally larger number of cycles, it is important to remember that in each kernel call, 1024 convolution points are being calculated at once. With this considered, each convolution point takes 694 cycles to compute, which is a 1312.24% speedup from the original kernel design. By utilizing more of the FPGA DRAM within the SSD device and utilizing Loop Unrolling, a more optimized kernel can be achieved. We still believe more kernel optimizations could be achieved inside the for loops, outside of parallelization or packing memory. This could reduce the kernel cycle time even further.

C. SmartSSD Application

The collected timing results for the read, computation, write, and total latency delays are shown in Tables II through V under Appendix A.

Figure 3 demonstrates the drastic overhead in data movement delays AND kernel computation time. The overall execution time of the original kernel and host code was a staggering 32.2476 seconds, over the baseline target 0.6623 seconds. This motivated us to explore alternate optimization techniques, which were referenced above under the Methodology section. It can be seen that marginal gains are made for each optimization except when adding multiple axi busses in parallel. The target SmartSSD application started to attain similar execution time after parallelizing the kernel computations, and packing more data into the FPGA DRAM, as seen by the drop when using 40 kernel points per call and less memory transactions in general, while still transmitting the same amount of data to/from the SSD and FPGA.

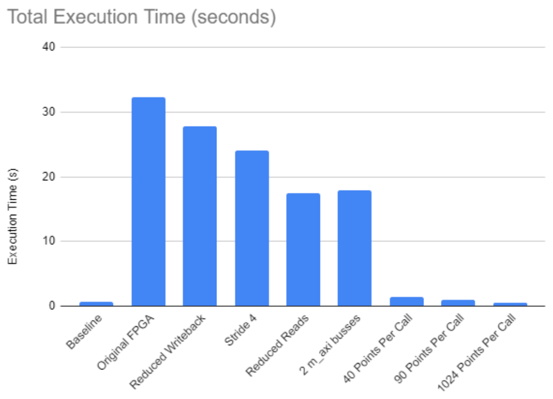


Figure 3: Total Layer Execution Time

To better visualize this improvement, we generated plots of the total execution time with the baseline CPU application and only the measurements with 40, 90, and 1024 kernel calls within the SmartSSD application, as referenced in Figure 4. A lower execution time over the baseline process was achieved only when we used the most packed memory calls we could, with 1024 convolution points in one kernel call.

The main benefit of collecting individual timing measurements for the read, compute, and write transactions was that we could analyze the speedup over the baseline for each kernel design, referenced in Figure 5. As expected from our timing results, only the final optimization with 1024 concurrent convolution points in a single kernel call achieved a higher speedup. Notably, the latency for the high bandwidth data reads significantly decreased, leading to a speedup of 174.87% for the final 1024 point kernel configuration. The final convolution kernel achieved a total speedup of 138.61%.

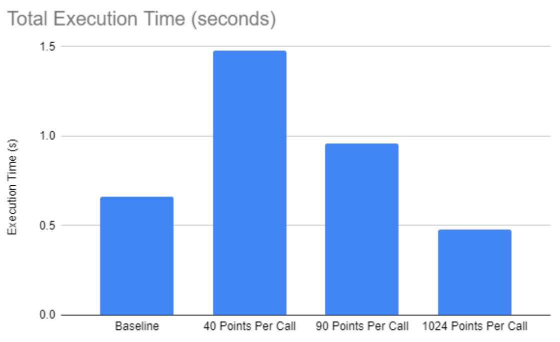


Figure 4: Total Layer Execution Time with Multiple Kernel Points

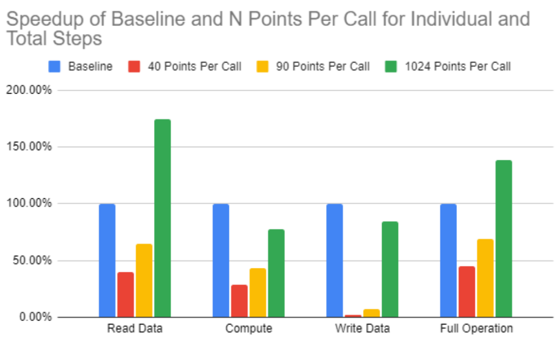


Figure 5: Speedup

Figure 6 displays the relative amount of time each application spent on computation versus data movement (read and write transactions). As the kernel becomes more packed, and the read/write latencies improve for the SmartSSD application, the percent of data movement reduces from the baseline 70.50% down to 53.82%, while achieving a faster overall execution time. This helps to reduce the gap in data movement delays, as referenced in related works. A tabular output of this data is represented in Appendix A under Table VII.

Corresponding plots for each individual latency for the total convolution layer are displayed in Figures 7 through 9 in Appendix A. Separate graphs for the read, compute, and write delays for the baseline and kernel optimizations with multiple convolution points are referenced in Appendix A in Figures 10 through 12.

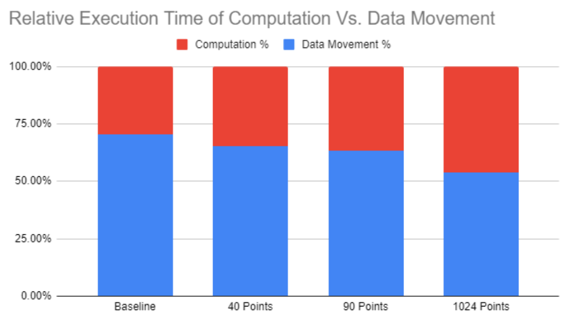


Figure 6: Relative Execution Time of Computation and Data Movement

VI. Analysis

In general, the collected results show that SmartSSD is a viable platform for an image inference application, specifically by integrating a high-bandwidth floating point multiply-accumulate module. While the computation time of the kernel was still less than the baseline CPU application, the speedup of the high-bandwidth read transactions for the FPGA from the SSD showed a wide margin for improvement. While the kernel still took slightly longer to compute, this does not mean that the kernel cannot be optimized further with more parallelization when computing a single convolution point, or even computing multiple convolution points in parallel at once. Since the host code is set up, it would be trivial to optimize this code within Vitis HLS for an improved target design.

One surprise from the results was that the write data delay from the FPGA kernels still took more time than the baseline code. This could be due to the delay for writing 1 block of 1024 entries back being too small to benefit from an improved speedup. Similar to the original designs, which read/wrote 5 1KB blocks per kernel call, it shows that low bandwidth usage with SmartSSD is less effective than many common CPU applications. We were surprised at the overhead with timing the host code control for activating the kernel, and reading/writing to the SSD with the pwrite and pread commands. By reducing the amount of these calls, and widening the bandwidth on the PCIe bus to the FPGA, a higher speedup can be achieved. Perhaps there is a way to store the resulting convolution points in the FPGA DRAM, and have a separate kernel write back every single convolution point at once, which could widen the block transfers in a similar fashion to the read data optimizations.

The most ideal result is that while speeding up the target design, the percent of time spent on data movement overall increased for every single kernel application with multiple convolution points, seen in Figure 6. While attempting to speed up the design, the amount of time spent on moving data to/from the FPGA was reduced when increasing the memory transaction bandwidths. By reducing the data movement delay, it opens more opportunity to optimize the target design on the FPGA kernel, which is open to many resources with Xilinx documentation online. Previously, the design would be limited by data movement, which could seldom be improved due to limititations of the common CPU structure and cache hierarchy.

VII. Future Work

With the host code and project structure with SmartSSD set up, improvements could be made such as optimizing the kernel for a lower latency, expanding the Host code interface for more applications, or adding improved timing analysis procedures. Other works have explored more modular approaches to compiler CNNs into RTL instead of a C compiler, which may be a great addition to this work [14]. This is only one layer of the 487 framework, and seeing it functional in SmartSSD provides a baseline for the implementation of other layers, with the proof that speedup can be achieved by reducing the latency of data movement. Regardless, through this work, we hope that we can help other students and graduate researchers with SmartSSD, regardless of the application.

VIII. Conclusion

Overall, the high bandwidth requirement for a neural network interface has provided a great example of a useful project to utilize SmartSSD. Through utilizing the given CPRE 487 neural network inferrence library, we have gained a large sum of knowledge about the Vitis toolflow and how SmartSSD can be utilized. Through developing this framework, we have learned more about Vitis HLS synthesis and what types of kernels would be optimal for the SmartSSD application. Alongside this, we have learned about potential ways to analyze I/O latency between the SmartSSD and FPGA through the use of Peer to Peer connections in Vitis, which allows direct read/write access between the SSD device and FPGA within SmartSSD, without the host needed. By utilizing SmartSSD we have shown benefits for high bandwidth image inferencing applications by reducing the data movement latency.

IX. Responsibilities

Shared Responsibilities

- Report Writing
- Presentation Development
- Results Analysis

Jake's Responsibilities

- SmartSSD Background Literature Search
- SmartSSD Bringup
- Vitis Research

William's Responsibilities

- Neural Network Background Literature Search
- Neural Network Host Bringup
- Neural Network Dataset Classification
- Other Layer Implementation Exploration + Implementation

References

[1] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Trans. Storage*, vol. 16, oct 2020.

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

[3] D. Fakhry, M. Abdelsalam, M. W. El-Kharashi, and M. Safar, "A review on computational storage devices and near memory computing for high performance applications," *Memories - Materials, Devices, Circuits and Systems*, vol. 4, p. 100051, 2023.

[4] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, 2020.

[5] N. Prakriya, Y. Yang, B. Mirzasoleiman, C.-J. Hsieh, and J. Cong, "Nessa: Near-storage data selection for accelerated machine learning training," in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '23, (New York, NY, USA), p. 8–15, Association for Computing Machinery, 2023.

[6] M. Soltaniyeh, V. L. Moutinho Dos Reis, M. Bryson, R. Martin, and S. Nagarakatte, "Near-storage acceleration of database query processing with smartssds," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 265–265, 2021.

[7] M. Soltaniyeh, V. Lagrange Moutinho Dos Reis, M. Bryson, X. Yao, R. P. Martin, and S. Nagarakatte, "Near-storage processing for solid state drive based recommendation inference with smartssds®," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, (New York, NY, USA), p. 177–186, Association for Computing Machinery, 2022.

[8] E. Bank Tavakoli, A. Beygi, and X. Yao, "Rpknn: An opencl-based fpga implementation of the dimensionality-reduced knn algorithm using random projection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 4, pp. 549–552, 2022.

[9] W. Qiao, J. Oh, L. Guo, M.-C. F. Chang, and J. Cong, "Fans: Fpga-accelerated near-storage sorting," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 106–114, 2021.

[10] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "Nascent: Near-storage acceleration of database sort on smartssd," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, (New York, NY, USA), p. 262–272, Association for Computing Machinery, 2021.

[11] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, "Nascent2: Generic near-storage sort accelerator for data analytics on smartssd," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, jan 2022.

[12] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, (New York, NY, USA), p. 1221–1230, Association for Computing Machinery, 2013.

[13] M. Soltaniyeh, V. L. Moutinho Dos Reis, M. Bryson, R. Martin, and S. Nagarakatte, "Near-storage acceleration of database query processing with smartssds," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 265–265, 2021.

[14] Y. Ma, N. Suda, Y. Cao, J.-s. Seo, and S. Vrudhula, "Scalable and modularized rtl compilation of convolutional neural networks onto fpga," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–8, 2016.

APPENDIX A

	Baseline	Original FPGA	Reduced Write	Stride 4	Reduced Read	2 m_axi busses	40 Points	90 Points	1024 Points
Point Time (us)	35.3864	173.0000	169.4000	159.4000	92.6000	91.0000	11.5750	7.1378	2.6463
Layer Time (s)	0.4644	17.3609	16.9996	15.9961	9.2926	9.1320	1.1616	0.7163	0.2656
Layer Difference (s)	0.0000	-16.8965	-16.5352	-15.5317	-8.8282	-8.6676	-0.6972	-0.2519	0.1988
Speedup	1.0000	0.0267	0.0273	0.0290	0.0500	0.0509	0.3998	0.6483	1.7487

Table II: SSD Read Latency

	Baseline	Original FPGA	Reduced Write	Stride 4	Reduced Read	2 m_axi busses	40 Points	90 Points	1024 Points
Point Time (us)	2.5044	263.6000	295.4000	248.2000	246.8000	211.8000	6.7900	4.5067	2.4986
Layer Time (s)	0.1953	26.4528	29.6440	24.9074	24.7669	21.2546	0.6814	0.4523	0.2507
Layer Difference (s)	0.0000	-26.2574	-29.4486	-24.7120	-24.5715	-21.0592	-0.4860	-0.2569	-0.0554
Speedup	1.0000	0.0074	0.0066	0.0078	0.0079	0.0092	0.2867	0.4319	0.7791

Table III: Single Point Kernel Computation Latency

	Baseline	Original FPGA	Reduced Write	Stride 4	Reduced Read	2 m_axi busses	40 Points	90 Points	1024 Points
Point Time (us)	0.0590	82.8000	35.2000	38.6000	49.8000	59.2000	1.3300	0.3378	0.0299
Layer Time (s)	0.0025	8.3091	3.5324	3.8736	4.9975	5.9408	0.1335	0.0339	0.0030
Layer Difference (s)	0.0000	-8.3066	-3.5299	-3.8711	-4.9950	-5.9383	-0.1309	-0.0314	-0.0005
Speedup	1.0000	0.0003	0.0007	0.0007	0.0005	0.0004	0.0190	0.0747	0.8440

Table IV: SSD Write Latency

	Baseline	Original FPGA	Reduced Write	Stride 4	Reduced Read	2 m_axi busses	40 Points	90 Points	1024 Points
Point Time (us)	6.5994	321.3449	277.1923	240.3659	174.4021	179.0099	14.7142	9.5404	4.7612
Layer Time (s)	0.6623	32.2476	27.8168	24.1212	17.5016	17.9640	1.4766	0.9574	0.4778
Layer Difference (s)	0.0000	-31.5853	-27.1545	-23.4589	-16.8393	-17.3017	-0.8143	-0.2951	0.1845
Layer Speedup %	1.0000	0.0205	0.0238	0.0275	0.0378	0.0369	0.4485	0.6917	1.3861

Table V: Total Single Point Latency

	Baseline	40 Points Per Call	90 Points Per Call	1024 Points Per Call
Read Data	100.00%	39.98%	64.83%	174.87%
Compute	100.00%	28.67%	43.19%	77.91%
Write Data	100.00%	1.90%	7.47%	84.40%
Full Operation	100.00%	44.85%	69.17%	138.61%

Table VI: Speedup of Multi-Point Kernel Designs

	Baseline	40 Points	90 Points	1024 Points
Data Movement %	70.50%	65.23%	63.54%	53.82%
Computation %	29.50%	34.77%	36.46%	46.18%

Table VII: Relative Data Movement Vs. Computation Time

Appendix B

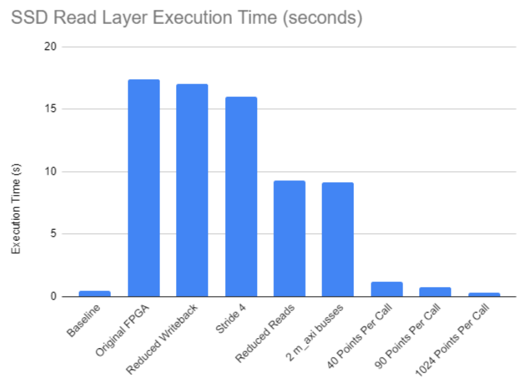


Figure 7: Read Layer Execution Time

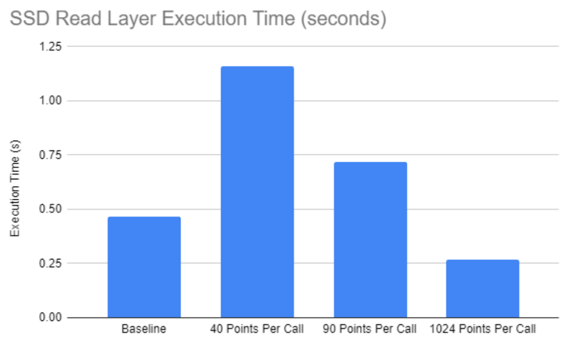


Figure 10: Read Layer Points Execution Time

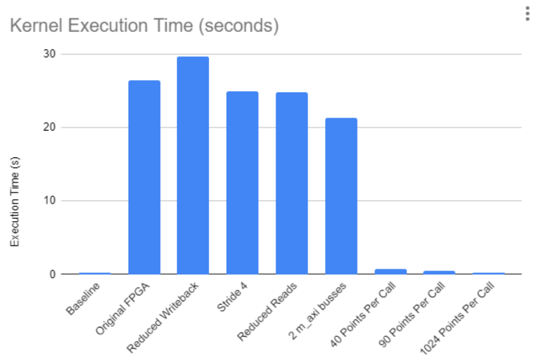


Figure 8: Compute Layer Execution Time

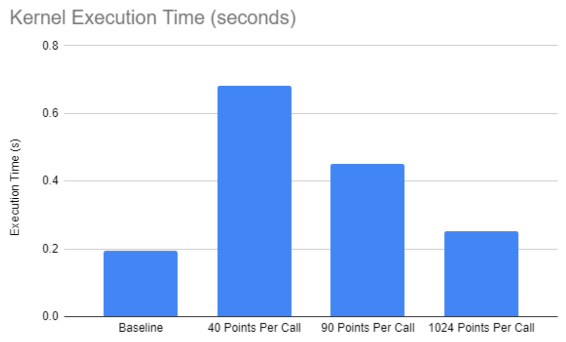


Figure 11: Compute Layer Points Execution Time

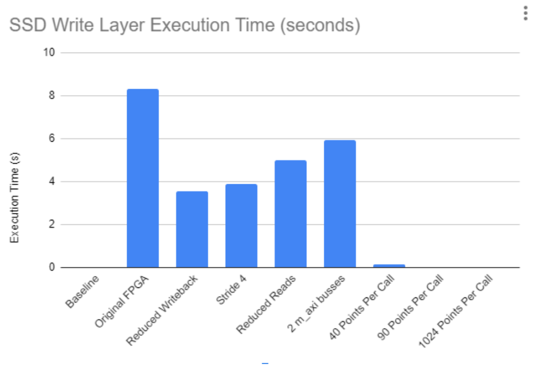


Figure 9: Write Layer Execution Time

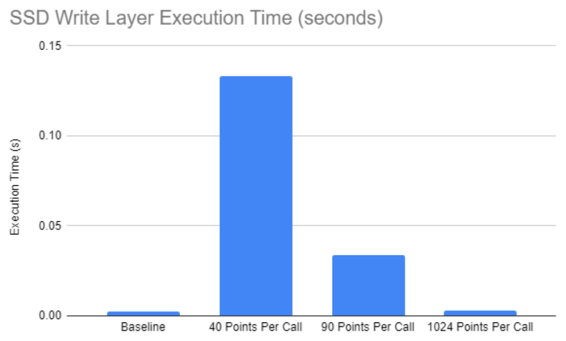


Figure 12: Write Layer Points Execution Time