2021 IEEE 39th International Conference on Computer Design (ICCD) | 978-1-6654-3219-1/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCD53106.2021.00084

HosNa: A DPC++ Benchmark Suite for Heterogeneous Architectures

Najmeh Nazari Bavarsad*, Hosein Mohammadi Makrani*, Hossein Sayadi†, Lawrence Landis‡, Setareh Rafatirad*, and Houman Homayoun*

*University of California Davis, †California State University Long Beach, ‡Intel

*{nnazaribavarsad, hmakrani, srafatirad, hhomayoun}@ucdavis.edu, †hossein.sayadi@csulb.edu, ‡lawrence.landis@intel.com

***Abstract*—Most data centers equipped their general-purpose processors with hardware accelerators to reduce power consumption and improve utilization. Hardware accelerators offer highly energy-efficient computation for a wide range of applications; however, their programming is not as efficient as processors. To bridge the gap, Intel developed a cloud-based infrastructure called DevCloud that connects Intel® Xeon® Scalable Processors to GPUs and FPGAs to deliver high compute performance for emerging workloads. DevCloud assists developers with their compute-intensive tasks and provides access to precompiled software optimized for Intel® architecture. To reduce programming complexity and minimize the barriers to adopt new innovative hardware technology, Intel also provided a unified, cross-architecture programming model called oneAPI based on the Data-Parallel C++ (DPC++) language. In this paper, we introduce HosNa, the first DPC++ benchmark suite that can be used for the evaluation of the Intel FPGAs and DPC++ productivity. Moreover, we present the characterization of proposed benchmarks and the evaluation of implemented hardware accelerators in terms of speedup and latency.**

I. Introduction

Recently, FPGA has become a promising solution for hardware acceleration. Designing hardware accelerators using RTL languages such as Verilog or VHDL is time-consuming and complex. To overcome this challenge, researchers developed High-Level Synthesis (HLS) tools that can automatically synthesize languages such as C or C++. However, HLS tools are able to generate hardware designs with sub-optimal performance. Recently, Intel used Data-Parallel C++ (DPC++) as a high-level language for data-parallel programming productivity. Developers can use DPC++ to easily port their code across different platforms such as CPUs, GPUs, and FPGAs.

Programming for FPGAs with DPC++ slightly differs from traditional software coding. DPC++ users usually need to apply different optimization techniques to meet their design and device constraints. There is a list of compiler optimization flags, attributes, pragma, and extensions that allow users to customize the DPC++ application compilation process. These features in combination with other common optimizations such as unrolling, pipelining [1] [2] [3], and memory partitioning[4] [5] makes it hard for a software developer to use easily DPC++. Having a standard benchmark suite that can easily enable the users to explore different optimization techniques and evaluate the hardware platform capabilities is required for DPC++ developers.

To the best of our knowledge, there is not any DPC++ based benchmark suite available in the community that can be used to systematically evaluate the optimization strategies and explore the design space of hardware acceleration using the DPC++ language. Hence, we introduce HosNa — the first open-source[1] suite of DPC++ benchmarks for Intel platform. Our suite includes common cryptography applications, image processing kernels as well as image rendering, machine learning, and a soft-core processor. HosNa can be employed to apply several optimization techniques provided by the oneAPI toolkit to meet the hardware design constraints and achieve optimal results.

HosNa can be used in tutorials for FPGA developers to get a hands-on Intel's platform. Researchers also can use our suite to evaluate the features and functionality of DPC++. Moreover, our benchmark can be used as a baseline for any further analysis of optimization strategies as we provided a simple, and easy to use DPC++ code.

The rest of this paper is organized as follows: In Section 2, we present related work on High Level Synthesize benchmarking and optimization strategies used; Section 3 presents background on DPC++. We introduce HosNa in Section 4 with full details; we show the experimental results in Section 5 and conclude our work in Section 6.

II. Related Work

FPGA programming currently differs significantly from the common practice of software programming, even with the use of HLS tools. Hardware developers have to explore various complex design trade-offs such as performance, power, area, and cost, instead of simply focusing on functional correctness and execution time. As developers cannot directly use traditional software benchmark suites to evaluate the performance of their hardware accelerator designs, researchers introduced several HLS benchmark suites. Intel started to use DPC++ in developing hardware accelerators, however, there is not any DPC++-specific benchmark suite available in the research community for evaluating various aspects of DPC++ and

[1] https://github.com/hoseinmmm/HosNa

978-1-6654-3219-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ICCD53106.2021.00084

Intel's platform. The closest works that we can consider as a baseline and convert them to DPC++ are HLS benchmark suites. CHStone [6] is one of the most popular C-based HLS benchmark suite, that includes applications from arithmetic, signal processing, and security domain. The other well-known suite is MachSuite [7], which provides different implementations for the same application to enable comparisons at the system level. One of the latest efforts is Rosetta [8] which is a realistic HLS benchmark suite for software programmable FPGAs and we adopted few benchmarks from it in this work.

HLS developers employed benchmarks from other communities to expand the available space for exploration. One example is Rodinia [9] that is designed to be used for GPU benchmarking [10] [11]. Moreover, there are OpenCL benchmarks that can be be executed on Intel FPGA platforms such as MetaCL [12] and Spector [13]. Spector is developed to be used for automatic design space exploration (DSE) and therefore, it covers large design spaces. The list of HLS benchmarks adopted from other domains is huge and it starts from Polybench [14] to [15], [16], [17], [18], [10], [19], and [20]. Rodinia [9] is another suite that provides benchmarks for heterogeneous computing by using CUDA, OMP, and OpenCL. The popular HLS suits are easy to use and are comprehensive, however, they lack the capabilities of DPC++ and new technologies of Intel's toolkit. There is only one work that adopts DPC++ and ports one Legacy CUDA Stencil source code to DPC++ using oneAPI toolkit [21]. By considering the need for a new benchmark suite to enable the performance evaluation of DPC++ and Intel's FPGAs, our work is the answer to such need.

III. Background on DPC++

DPC++ language is an extension of the C++ language for data-parallel programming with complete support for SYCL. SYCL is an industry-standard that allows using OpenCL to add data parallelism to C++ for the heterogeneous platform. DPC++ has all SYCL features and also adds some new features. It also has extra capabilities that led to simplifying programmability and optimization compared to openCL. Therefore, DPC++ as a high-level language presents both modern C++ benefits and the performance benefit of parallelism in heterogeneous systems. Developers can exploit high performance from different hardware consists of general-purpose processors (CPUs), GPUs, and FPGAs by using DPC++. Moreover, DPC++ offers functional portability, and hence, the programmer can reuse code through the mentioned architecture. In the Intel platform, DPC++ is part of OneAPI, a unified cross-architecture programming model that decreases programming complexity.

DPC++ features support both data-parallel programming and task-parallel programming. DPC++ applications include a combination of both host code and device code. Host code that is executed by the CPU determines and controls the execution of computation on available devices. For instance, data movement and compute dispatching to devices is performed by the host code. Moreover, it can do compute-intensive work that is native C++ code. Device code runs on the accelerators or processors (not the processor that executes host code) to achieve high compute performance for applications. Host code submits works to devices through queues. Host code and device code can be executed asynchronously. However, for many applications, host code and device code must be run in a specific order to obtain the correct result, meaning there are dependencies between tasks.

In DPC++, both host code and device code can exist in the same source file, and hence, the compiler can easily optimize both codes together. Managing memory can be performed explicitly by the developer or implicitly at the runtime. DPC++ offers three abstractions to manage memory consists of Unified Shared Memory (USM)(pointer-based approach), buffers (represented by buffer template class), and images (special type appropriate for image processing).

IV. Benchmarks Description

In this section, we discuss our application selection in detail, and we briefly introduce each benchmark's functionality. We chose workloads to satisfy two criteria of coverage and diversity. Coverage shows the representativeness of our workloads with respect to the field. It is important to have at least one kernel similar to any application a given user run. On the other hand, diversity is also significant, since each kernel must bring a new behavior to our list. To achieve our criteria, we surveyed the literature and directly adopted applications to ensure that our benchmark suite contains applications the community cares about them. Our benchmarks will provide new targets for accelerator designers and system architects. Currently, we released the first and basic version of our benchmark suit. The target of current version is to be used in tutorials and can be considered as a baseline. We will expand the number of benchmarks in near future and eventually we will release different versions of our benchmark for each domain.

We have considered seven realistic benchmarks in various fields consist of machine learning, cryptography, image processing, and high-performance computing. For each design, we provide the software version and the optimized DPC++ version that can be executed on both FPGA (Arria 10) and GPU on DevCloud.

We select both compute-intensive and memory-intensive benchmarks to cover a wide range of applications. Our applications have diverse sources of parallelism, such as instruction-level parallelism (ILP) and task-level parallelism (TLP). To adopt HLS benchmarks, our DPC++ designs must be customized using a variety of optimization techniques. For this purpose, we used different techniques such avoiding pointer aliasing, avoiding pointer arithmetic, converting nested loops to a single loops where it was possible, global memory access optimization by preloading data into local memory, having a continuous memory access whenever possible, specifying the number of memory banks and bank width, avoiding to define a float variable as much as possible, tree balancing for floating-point optimization, and removing floating-point rounding operation. Moreover, we used pragmas and attributes for loops

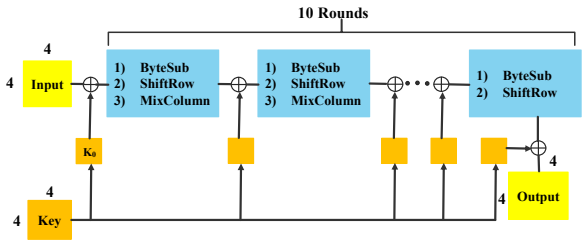

Fig. 1: Internal structure of AES

Fig. 2: Example of convolution on an image

and memory such as loop unrolling and max concurrency in the source code.

A. AES Cipher

The Advanced Encryption Standard (AES) [22] invented by cryptographers Joan Daemen and Vincent Rijmen is an iterated block cipher deployed instead of the DES algorithm. We implemented this benchmark from scratch and the design is as follow: AES is a symmetric block cipher, meaning it uses the same key for both data encryption and data decryption. AES gives a 16-byte state matrix (128-bit data as a grid of 4×4 matrix) and a choice of three keys (128, 192, and 256) as an input and then encrypt them through a series of alternating computations called round function. Each round function comprises the substitution phase, permutation phase, and adding round key. Note that the key length determines the number of internal rounds of the block cipher. The number of rounds is 10, 12, and 14 for AES-128, AES-192, and AES256, respectively. Since the most common key size used is 128, we implement AES-128 in this paper shown in Figure 1.

The AES algorithm starts with adding a round key step and is then followed by 10 rounds. Each round is consists of four-step: 1. Substitute bytes 2. Shift rows 3. Mix Columns 4. Add Round Key. The First step (Substitute bytes) is known as the substitution phase, and both the second and third steps (Shift rows + Mix Columns) are known as the permutation phase. Substitute bytes function known as SubByte maps each element of inputs matrix(4×4) to the new value using s-box that is a 16×16 matrix of bytes. This step can be implemented by the lookup table. The SubByte output (matrix 4×4) feeds to the ShiftRow function as an input. In this step, a circular left shift is done on the rows of the matrix. The first row, second row, third row, and the fourth row are shifted 0 Byte, 1 Byte, 2 Byte, and 3 Byte to the left, respectively. In the Mix Column stage, the input matrix is multiplied with a constant matrix. In the last step (AddRoundKey function), the 128-bit input matrix is XORed with the 128 bit unique round key. Note that the Mix Column step is omitted in the last round.

B. 2D Convolution

Convolution is a mathematical operation on two signals that produces a third signal which expresses how the shape of one is modified by the other [23]. 1D convolution is referred to as convolution involving one-dimensional signals. The 2D convolution is the heart of image processing where the first signal is the image, and the other signal is a kernel, which is simply a small matrix of weights. This kernel moves over the 2D input data, performing a dot product with the part of the input it is currently on, and summing them up. The output of the operation is assigned to the cell in the convolution array for which the convolved array is seen. This benchmark has been implemented by our group and 2D convolution is frequently deployed in convolutional neural networks (CNNs) used for classification tasks in the machine learning domain.

We denote $C = K * I$ as the convolution of I by K. The dimension of the K (of size (2N+1) x (2N+1), where N represents an integer number) is small relative to the I (an image of size 128 x 128). It is possible to perform the design space exploration with the same design by changing the N and implement different kernels by simply changing the values of the K coefficient table.

We implemented the following example presented in Figure 2 that shows the result of calculating the dot product for the operator K = (-1, -1, -1; -1, 8, -1; -1, -1, -1) against the image. This Kernel detects image outlines.

C. Digit Recognition

This application has been adopted from Rosetta [8]. Digit recognition classifies the handwritten digits as 10 digits (0–9) from the famous MNIST dataset with 18000 training samples and 2000 test samples through the K-Nearest-Neighbor (KNN) algorithm. This application downsamples each image of the handwritten digit to 14x14 that each pixel is just one bit. Therefore, Each image is 196-bit unsigned integer.

The KNN algorithm is done in three steps. 1) Calculating Hamming distance: It computes the distance between a test input and each training image. 2) Find the closest neighbors: It finds the labels of the training images with the K shortest distances 3) Vote for labels: It votes among K labels to take the majority vote and determine the label of test input. The first step can be done through bitwise XOR on the inputs(test input and training image), and then, the XOR results are fed to the population count to obtain hamming distance. We can exploit both bit-level and data-level parallelism in the first step. The second step examines the Hamming distance list to find K's shortest distance. Finally, the voting step determines the most frequent label. This step compares and sorts integer numbers, and hence, exploiting parallelism is more difficult compared to the first step.

The amount of communication compared to the amount of computation is negligible in digit recognition applications.

Moreover, both training images and their labels have been stored in on-chip memory. Therefore, Digit recognition is a compute-intensive application rather than a memory-intensive.

D. Spam Filtering

This design trains a logistic regression model to classify emails as either spam or normal. We use stochastic gradient descent (SGD) in our accelerator to train the model. HosNa's version is adapted from a Rosseta [8]. Each email is a 1024 dimensional vector (word frequencies stored as 16-bit fixed-point numbers). We also use five training epochs but with a minibatch size of 100; This is different with Rosetta as we needed to fit our design in Arria 10.

E. MIPS

We adopted a Simplified MIPS processor from CHstone [24]. MIPS is a reduced instruction set computer (RISC) instruction set architecture (ISA). The implemented CPU has 30 instructions, and a sorting application is served as test vectors.

F. Hough Transform

The Hough transform is used in computer vision applications. After an image has been processed with an edge-detection algorithm such as a Sobel filter, the image is left in monochrome (black/white). It is suitable for many further detection algorithms to consider the image as a set of lines. However, an image of black and white pixels is not an appropriate representation of these lines to algorithms such as object detection. The Hough transform is a transform from pixels to a set of line votes.

G. 3D Rendering

This design is another adoption from Rosetta [8] and renders 2D images from 3D models (3D triangle mesh). It is composed of the following stages:

- Projection: 3D triangle to 2D triangle
- Rasterization (1, 2): search pixels in the 2D triangle within the bounding box
- Z-culling: hide or display pixels according to each pixel's "z" value (depth)
- ColoringFB: coloring framebuffer according to the zbuffer

Two major optimizations have been applied to this application. First Data Packing optimization for inputs. Each 3D triangle is packed into a 32-bit integer for faster off-chip data transfer. The second is Coarse-Grained pipelining.

V. Experimental Results

In this section, we introduce the platform of Intel's DevCloud, and then we present a full characterization of our selected benchmarks. Then we provide the FPGA implementation and exploration results.

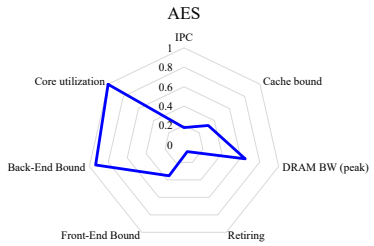

Fig. 3: Micro-Architectural break-down of AES

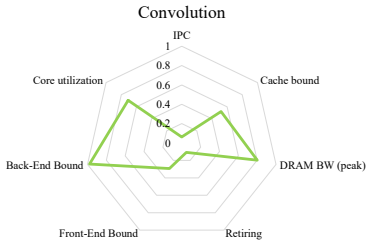

Fig. 4: Micro-Architectural break-down of Convolution

A. Environment

Intel DevCloud is a cluster of Intel® Xeon® Scalable Processors connected to GPUs and FPGAs. However, taking advantage of multiple types of architectures can be a challenge for developers as separate tools are required for each architecture, and code reuse is limited. Precompiled optimized software is available on DevCloud for Intel's platform. To reduce programming complexity Intel provided a cross-architecture programming model called oneAPI. The current processor available on DevCloud is Intel(R) Xeon(R) Gold 6128 CPU running at 3.40GHz and the FPGA is Arria 10 GX Platform Accelerator Card with 20 nm technology (Part number: 10AX115S2F45I2SGES). Applications are running on Ubuntu 18.04.3 with the 4.15.18 Linux kernel. This FPGA has 1.15 million logic elements, over 5 million memory bits, 1518 DSP blocks, and 2713 RAM blocks. It should be noted that Arria 10 GX is connected to a small PCI Express card and is more suited for backtesting, simple database acceleration and image processing, or a task with a lightweight artificial intelligence (AI). For more compute-intensive tasks, a system equipped with Stratix FPGA is a better solution.

B. Characterization

Figures 3, 4, 5, 6, 7, 8, and 9 show the characterization of HosNa benchmarks running solely on the CPU. Based on Top-Down methodology [25], applications can be classified into

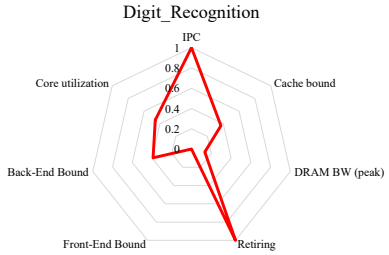

Fig. 5: Micro-Architectural break-down of Digit Recognition

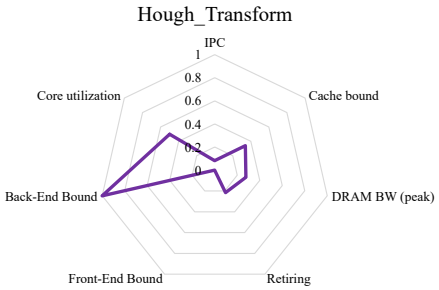

Fig. 6: Micro-Architectural break-down of Hough Transform

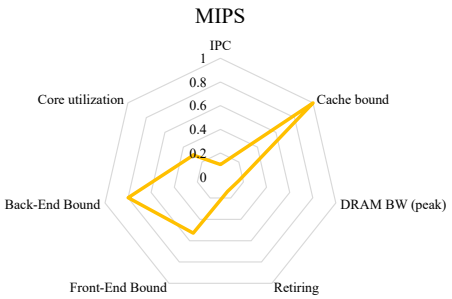

Fig. 7: Micro-Architectural break-down of MIPS processor

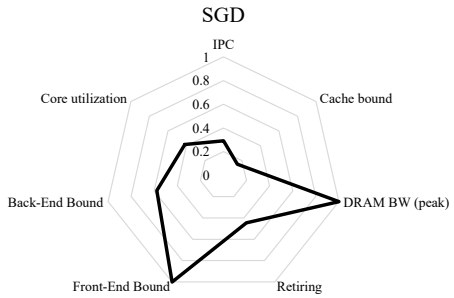

Fig. 8: Micro-Architectural break-down of Spam filtering

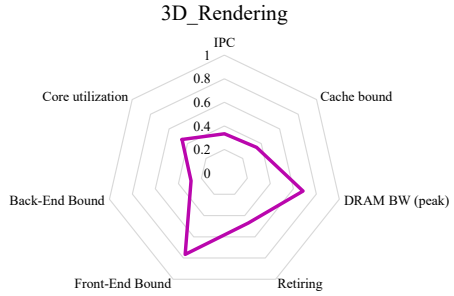

Fig. 9: Micro-Architectural break-down of 3D-Rendering

three categories such as I/O bound, core bound, and memory bound. Top-Down methodology chooses the micro-op (µop) queue of a out-of-order server as a dividing point between a core's front-end and back-end, and uses it to classify µop pipeline slots in four broad categories: Retiring, Front-end bound, Bad speculation, Back-end bound. Out of these, The Retiring classifies as "useful work" and the rest prevent the workload from utilizing the full core width. We normalized these metrics based on the maximum value to show them in the Radar chart.

Moreover, we present the IPC (instructions per cycle). This metric shows how many instructions a CPU can execute in one cycle. The DRAM bandwidth peak usage is also presented for each application. There is another metric called Cache bound that indicates the proportion of cycles are being spent on data fetches from caches. We also present the physical core utilization. All the information has been extracted from Intel VTune. The characterization shows that our benchmarks' behavior is diverse.

The characterization results for AES shows that the IPC is low compared to other benchamrks. This is caused by issues such as memory stalls. The Back-End result shows a significant portion of pipeline slots are remaining empty. AES is 32% Cache Bound and several cycles are being spent on data fetches from caches. Convolution has similar behavior to AES. However, digit recognition has the highest IPC and the results indicate cache hit rate is high and it reduces access to main memory. This behavior leads to an increase in the percent of retiring instructions compared to AES and convolution. Hough transform is as Back-End bound with low IPC which is the indication of poor performance.

Intel VTUne reported that MIPS is 72.9% core bound and this metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it indicates the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in the program's data- or instruction- flow are limiting the performance. Usually, Long-latency operations like divides and memory operations can cause this, as can too many operations being directed to a single execution port. The report also shows 69.1% of Clockticks were used only for serializing operations on execution ports. Unlike MIPS, SGD is memory intensive. The report shows that 75.2% of Clockticks were L1 Bound. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and stores. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1. 3D rendering has a high retiring rate (47.1% of Pipeline Slots). It shows a high fraction of pipeline slots were utilized by useful work. Usually, the goal is to make this metric value as big as possible. Moreover, this application has high peak DRAM bandwidth usage.

C. Common optimization techniques

In this section, we provide common FPGA optimization techniques applied to the benchmarks with an example on Hough Transform benchmark. In this example, we provided two snapshot of source codes for non-optimized and fully-optimized versions of the kernel. Figure 10 shows the non-optimized and Figure 11 shows the optimized version of the kernel.

```
cgh.single_task<class Hough_transform_kernel>([=]() {
  for (uint y=0; y<HEIGHT; y++) {
    for (uint x=0; x<WIDTH; x++){
      unsigned short int increment = 0;
      if (_pixels[(WIDTH*y)+x] != 0) {
        increment = 1;
      } else {
        increment = 0;
      }
      for (int theta=0; theta<THETAS; theta++){
        int rho = x*_cos_table[theta] + y*_sin_table[theta];
        _accumulators[(THETAS*(rho+RHOS))+theta] += increment;
      }
    }
  }

});
```

Fig. 10: Code snapshot of non-optimized kernel for Hough Transform application

```
cgh.single_task<class Hough_transform_kernel>([=]()
  [[intel::kernel_args_restrict]] {

  //Load from global to local memory
  [[intel::numbanks(256)]]
  short accum_local[RHOS*2][256];
  for (int i = 0; i < RHOS*2; i++) {
    for (int j=0; j<THETAS; j++) {
      accum_local[i][j] = 0;
     }
  }
  for (uint y=0; y<HEIGHT; y++) {
    for (uint x=0; x<WIDTH; x++) {
      unsigned short int increment = 0;
      if (_pixels[(WIDTH*y)+x] != 0) {
        increment = 1;
      } else {
        increment = 0;
      }

      #pragma unroll 32
      [[intel::ivdep]]
      for (int theta=0; theta<THETAS; theta++){
        int rho = x*_cos_table[theta] + y*_sin_table[theta];
        accum_local[rho+RHOS][theta] += increment;
      }
    }
  }
  //Store from local to global memory
  for (int i = 0; i < RHOS*2; i++) {
    for (int j=0; j<THETAS; j++) {
       _accumulators[i*THETAS+j] = accum_local[i][j];
    }
  }

});
```

Fig. 11: Code snapshot of optimized kernel for Hough Transform application

1) Avoiding Aliasing of Kernel Arguments: Due to pointer aliasing, the compiler must be conservative about optimizations that reorder, parallelize, or overlap operations that could alias. We apply the DPC++ [[intel::kernel_args_restrict]] kernel attribute any time we can guarantee that kernel arguments do not alias. This attribute enables more aggressive compiler optimizations and often improves kernel performance on FPGA. OpenCL programmers may recognize this concept as the restrict keyword. In Figure 11, code in line 2 signals to the compiler that there is no pointer aliasing.

2) Local Memory (Loop Optimization): Sometimes, the kernel cannot retrieve data fast enough from the memory (global memory). Hence, the algorithm is stalled by more than one clock cycle to input more data. This increases the initiation interval (II) thereby affecting overall performance. A solution is to transfer global memory contents to local memory before working on the data. In the optimized code, code under the "//Load from global to local memory" and "//Store from local to global memory" comments presents the transfer of data from global to local and vice-versa.

3) Unroll (Loop Optimization): We use the loop unrolling mechanism to increase program parallelism by duplicating the compute logic within a loop. The number of times the loop logic is duplicated is called the unroll factor. Depending on whether the unroll factor is equal to the number of loop iterations or not, we can categorize loop unroll methods as full-loop unrolling and partial-loop unrolling. In Figure 11, "#pragma unroll 32" statement shows that the computation loop being duplicated(32x) and parallelized.

4) Banking: For each private or local array in the DPC++ FPGA device code, the Intel oneAPI DPC++ Compiler creates a custom memory system in the program's datapath to contain the contents of that array. Memory attributes are a set of DPC++ extensions for FPGAs that enable the programmer to override the compiler's internal heuristics and to control the architecture of kernel memory. One of these attributes or techniques is known as banking.

Banks are structures that have independent ports from the rest of the memory structure, but that only contain a portion of the contents. For example, if we created two banks, one bank contains half of the data and the other bank contains the other half of the data, each half can be read independently. Specifying the numbanks (N) and bankwidth (M) memory attributes allow users to configure the local memory banks for parallel memory accesses. The banking geometry described by these attributes determines which elements of the local memory system our kernel can access in parallel. In optimized code presented in Figure 11, the "[[intelfpga::numbanks(256)]]" statement shows that the local cache memory is divided into 256 banks.

D. FPGA implementation results

We compiled each kernel to generate an FPGA hardware image. We extracted all the data from Intel oneAPI toolkit's report to present them in this section. Then, we executed the application on a machine equipped with Intel PAC. The results of running benchmarks are presented in Figure 13. This Figure presents the kernel execution time running on FPGA. Moreover, we presented the latency or the static schedule cycles of the kernel reported by Intel Quartus Prime. This latency is important for the throughput and the performance of the kernel. The lower Latency means higher throughput. The interesting observation is that we could not find a direct relationship between the kernel time and kernel schedule

TABLE I: Resource utilization report on Arria 10

	ALUTs	FFs	RAMs	MLABs	DSPs	Fmax (MHz)
Board Interface	179950 (21%)	358572 (21%)	492 (18%)	0 (0%)	123 (8%)	–
AES	128789 (15%)	26962 (2%)	106 (4%)	66 (0%)	0 (0%)	227
Convolution	19240 (2%)	30175 (2%)	1098 (40%)	61 (0%)	108 (7%)	196
Digit_Recognition	626963 (73%)	1720792 (101%)	315 (12%)	47783 (112%)	0 (0%)	–
Hough_Transform	45449 (5%)	59168 (3%)	1147 (42%)	364 (1%)	64 (4%)	236
MIPS	28513 (3%)	23052 (1%)	93 (3%)	102 (0%)	4 (0%)	177
SGD	112540 (13%)	238499 (14%)	240 (9%)	1783 (4%)	4 (0%)	148
3D_Rendering	165829 (19%)	199494 (12%)	181 (7%)	150 (0%)	11 (1%)	–

TABLE II: Resource utilization report on Stratix 10

	ALUTs	FFs	RAMs	MLABs	DSPs	Fmax (MHz)
Board Interface	466792 (25%)	928428 (25%)	3039 (26%)	0 (0%)	1291 (22%)	-
AES	135889 (7%)	50389 (1%)	201 (2%)	8 (0%)	0 (0%)	362
Convolution	25642 (1%)	44298 (1%)	2128 (18%)	19 (0%)	112 (2%)	470
Digit_Recognition	669468 (31%)	1809589 (40%)	486 (3%)	48465 (72%)	1 (0%)	315
Hough_Transform	63888 (3%)	87725 (2%)	928 (8%)	149 (0%)	66 (1%)	464
MIPS	48094 (3%)	50832 (1%)	238 (2%)	14 (0%)	4 (0%)	200
SGD	186594 (10%)	333980 (9%)	1814 (15%)	3452 (4%)	6 (0%)	387
3D_Rendering	174048 (9%)	231672 (6%)	229 (2%)	171 (0%)	10 (0%)	226

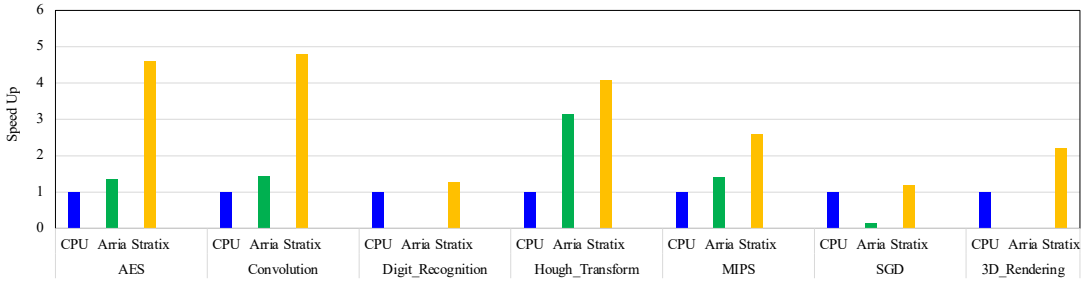

Fig. 12: Speedup of benchmarks on Arria and Stratix normalized to CPU

cycles. Our expectation was that a kernel with higher schedule cycles would have higher kernel execution time. However, by analyzing the Kernel graph provided by oneAPI toolkit, we found that the type of memory access inside the kernel is the most important parameter to impact the kernel execution time.

Table I shows the resource utilization results and the maximum frequency achieved by Intel Quartus Fitter. In this table, MLAB stands for memory logic array blocks. The first row of the table presents the static portion of the hardware accelerator which is the board interface and it is generated automatically and is a common component in all applications. In our implementation, we could not fit digit recognition in Arria 10. Therefore there is no result for the F_{max} and speed up of digit recognition application. For the 3D rendering, we faced an issue with the Intel Quartus Prime V. 20.1 available on DevCloud and as there was not a workaround for the successful compilation, we could not provide the implementation results. Therefore, we used the emulation report to fill the table for 3D rendering application. Table II shows the same result for Stratix 10 PAC. On Stratix 10, the amount of parallelism is increased and we achieved 1.76x performance benefits on Average.

We compared the performance of the accelerated application on FPGA with the performance of application running on a CPU without any accelerator, and Figure 12 shows the Speed Up provided by the FPGA accelerators. Based on the results, only SGD on Arria 10 is not able to beat the CPU due to several of-chip memory accesses and low-frequency operation.

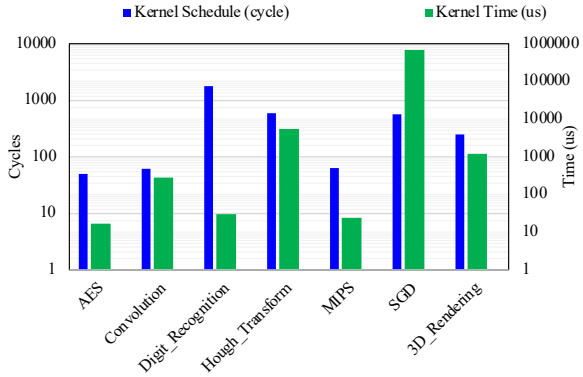

Fig. 13: Static schedule cycles of kernels VS kernels' time

However, SGD on Stratix 10 marginally outperforms the CPU.

E. Limitation

In this work, we decided to consider simplicity over optimality in our DPC++ implementation of applications, since HosNa is the first DPC++ benchmark suite and everyone should be able to use it for exploring the design space. Secondly, most of the time, an optimal design is not portable and compatible with all hardware platforms. We expect our benchmarks get examined and enhanced, to be able to address new challenges.

HLS tools mostly use directives to optimize the design [26], [27]. We also use Intel's directives to produce efficient hardware. However, we never claim that our design is opti-

mal. Performance is dependent on several factors and design parameters; therefore, a single set of directives is not always optimal. Our work, similar to prior works, is not without limitations. The one major issue is the size of the inputs to the benchmarks. Compared to real-world applications, our inputs are small. Therefore, to stress the design, users must generate new input sets in a case that they want to provoke meaningful interactions. HosNa currently supports parameters that can be changed to tune the applications such as maximum concurrency, memory access and memory type, data types, and input size. However, In the current version, only a subset of parameters is changeable For each benchmark. HosNa is developed to help mostly datapath design; we will continue the work to scale HosNa's input size and extend the applications in a way that a wider range of researchers can exploit it.

VI. Conclusion

We provided HosNa, the first open-source DPC++-based benchmark suite to enable a standard and measurable common ground for accelerator-centric research. We amortized the burden of manually implementing hardware accelerators to provide access to a diverse set of DPC++ hardware accelerators for researchers. All HosNa applications are ready to be executed on Intel DevCloud platforms (CPU, and FPGA). The diverse characteristics of HosNa benchmarks open the opportunity to explore customization strategies in the memory subsystem and computation part. DPC++ developers can use HosNa to exercise new optimization techniques to implement more complex programs and designs. We will continuously improve HosNa in the future and we will expand HosNa to include more emerging applications.

References

[1] S. Dai, R. Zhao, G. Liu, S. Srinath, U. Gupta, C. Batten, and Z. Zhang, "Dynamic hazard resolution for pipelining irregular loops in high-level synthesis," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 189–194.

[2] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang, "Architecture and synthesis for area-efficient pipelining of irregular loop nests," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1817–1830, 2017.

[3] Z. Zhang and B. Liu, "Sdc-based modulo scheduling for pipeline synthesis," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2013, pp. 211–218.

[4] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2014, pp. 199–208.

[5] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017, pp. 179–188.

[6] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the chstone benchmark program suite for practical c-based high-level synthesis," *Journal of Information Processing*, vol. 17, pp. 242–254, 2009.

[7] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 110–119.

[8] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez *et al.*, "Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 269–278.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[10] S. Wang, Y. Liang, and W. Zhang, "Flexcl: An analytical performance model for opencl workloads on flexible fpgas," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2017, pp. 1–6.

[11] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing opencl applications on fpgas," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2016, pp. 114–125.

[12] P. Sathre, A. Gondhalekar, M. Hassan, and W.-c. Feng, "Metacl: Automated "meta" opencl code generation for high-level synthesis on fpga," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–8.

[13] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An opencl fpga benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 141–148.

[14] T. Yuki and L.-N. Pouchet, "Polybench 4.0," 2015.

[15] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, 2013, pp. 29–38.

[16] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[17] W. Zuo, P. Li, D. Chen, L.-N. Pouchet, S. Zhong, and J. Cong, "Improving polyhedral code generation for high-level synthesis," in *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2013, pp. 1–10.

[18] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 97–108.

[19] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 430–437.

[20] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: a high-level performance analysis tool for fpga-based accelerators," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[21] S. Christgau and T. Steinke, "Porting a legacy cuda stencil code to oneapi," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2020, pp. 359–367.

[22] V. Rijmen and J. Daemen, "Advanced encryption standard," *Proceedings of Federal Information Processing Standards Publications, National Institute of Standards and Technology*, pp. 19–22, 2001.

[23] N. Nazari and M. E.Salehi, *Hardware Architectures for Deep Learning*. Materials, Circuits and Device, 2020, ch. Binary Neural Networks, pp. 95–115.

[24] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 2008, pp. 1192–1195.

[25] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 35–44.

[26] H. M. Makrani, F. Farahmand, H. Sayadi, S. Bondi, S. M. P. Dinakarrao, H. Homayoun, and S. Rafatirad, "Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019, pp. 397–403.

[27] H. M. Makrani, H. Sayadi, T. Mohsenin, S. Rafatirad, A. Sasan, and H. Homayoun, "Xppe: cross-platform performance estimation of hardware accelerators using machine learning," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019, pp. 727–732.