Characterizing Loop Acceleration in Heterogeneous Computing

Saman Biookaghazadeh
Arizona State University

Fengbo Ren
Arizona State University

Ming Zhao
Arizona State University

***Abstract*—Computation intensive applications usually consist of multiple nested or flattened loops. These loops are the main building blocks of the applications and embody a specific type of execution pattern. In order to reduce the running time of the loops, developers need to analyze the loops in the code and try to parallelize them on hardware accelerators, such as GPUs, TPUs, and FPGAs, which are increasingly available in the cloud. Unfortunately, the lack of understanding of loop characteristics and the ability of hardware accelerators in handling these types of loops prevents developers from choosing the right platform to develop their applications in the cloud. Also, developing and optimizing code for a specific accelerator is a time-consuming effort. To address these issues, this paper studies the effectiveness of different processors in accelerating common patterns of loops. It identifies five important types of loops that commonly exist in real-world applications, and presents *Loopy*, the implementations of these loops optimized for different architectures. Using Loopy, the paper also evaluates different hardware in accelerating the loop patterns. The result reveals the architectural differences among different accelerators with regard to different loop patterns. It also provides insights for the developers to choose the right accelerators for their applications. The current version of Loopy supports both FPGAs and GPUs, which are the most versatile and available accelerators.**

I. Introduction

Many applications can benefit from computing on hardware accelerators, ranging from cloud computing to big-data and edge computing. Examples of these applications include (1) analysis of large quantity of data on big-data platforms, (2) training and running artificial intelligence (AI) and machine learning models in the cloud, (3) processing streams of requests and data from IoT devices, and (4) modeling and simulating the behaviors of scientific applications. By using accelerators, applications can achieve higher throughput [1], lower response time [2], and/or lower energy consumption [3].

Cloud is by nature a heterogeneous computing environment with different types of accelerators available a a service (e.g., Google Cloud GPUs and TPUs [4], [5], Azure GPU VMs and FPGAs [6], [7], AWS GPU and F1 instances [8], [9]). These accelerators come with different capabilities and limitations. For example, FPGAs can be reconfigured to run any applications but can provide only low clock frequency; GPUs can be programmed using high-level languages to accelerate highly parallel applications; and TPUs are specifically designed for deep learning workloads. Although a general understanding of different accelerators is available, choosing the right accelerators for applications in a heterogeneous computing system is still a difficult problem.

Several related works have studied the performance of common algorithms on accelerators. For example, Rodinia benchmark and its follow-up work [10] are designed to benchmark heterogeneous platforms including CPUs, GPU, and FPGAs. These benchmarks usually provide insights on a macro level, for a complete algorithm on a hardware platform. However, they lack a thorough analysis of micro-level execution patterns that exist in different applications and the effectiveness of different hardware architectures in handling these patterns.

To address the above challenges, we study how the accelerators with different hardware architectures can accelerate different types of loops, which are the basic building blocks of almost every computationally intensive application. These applications typically consist of one or many nested and flattened loops. These loops can embody different patterns in terms of types and degrees of dependency and concurrency, and they can be found in many applications. For example, dynamic programming algorithms consist of one or more nested loops, where every iteration depends on another iteration that points diagonally in the iteration space. Therefore, abstracting the common loop patterns from applications and understanding how they perform on various hardware accelerators are essential steps towards optimally utilizing the accelerators for executing different applications. Although there is a great body of existing works on loop optimizations [11]–[24], they cannot provide cross-accelerator comparisons that can help developers choose the right platform for their applications in a heterogeneous computing system.

To support the study of loop accelerations across different platforms, we developed *Loopy*, a collection of five fine-grained loop patterns that commonly exist in real-world applications such as linear algebra, optimization, and data analytics algorithms. Loopy parameterizes the key aspects of these loop patterns, including the type and degree of dependencies, data bit-precision, operational intensity, and size of the iteration spaces. It allows them to be flexibly tuned to model diverse loop characteristics. Loopy provides optimized OpenCL implementations of these loop patterns for both GPU and FPGA, the two most versatile and available accelerators. We focus on OpenCL because it is an important framework for the emerging heterogeneous computing paradigm.

Based on Loopy, we evaluated the performance of important loop patterns on several typical accelerators, including Intel A10 FPGAs and Nvidia T4 and RTX2080 GPUs. Our study made several key findings. First, for three out of five loop

2159-6190/21/$31.00 ©2021 IEEE
DOI 10.1109/CLOUD53861.2021.00059

dependency patterns (intra-dimension dependency, conditional dependency, and half-parallelism half-dependency), FPGA has the potential to outperform GPU. For example, for the intra-dimension dependency pattern, the evaluated FPGA outperforms GPU by 17.5x. Second, for various computational intensities, FPGA can maintain an identical performance, whereas GPU performance is highly variable. For example, having eight conditional statements can degrade the GPU performance by up to 45%. Third, increasing the input data size can increase the performance difference between these two accelerators. For example, for the diagonal dependency loop pattern, the performance gap increases by 51%, while the input data size increases from 4MB to 256MB.

In summary, the contributions of this study include:

- identification and classification of common loop patterns in computationally intensive applications,
- optimization of these loop patterns on the OpenCL-enabled FPGAs,
- experimental analysis of the acceleration potentials of these loop patterns on GPUs and FPGAs, with regard to key configuration parameters, such as computational intensity, dependency and concurrency degrees, and input data size.

II. Background

A. Accelerators

Acceleration is becoming a critical de-facto for many computationally intensive workloads on various computing systems. While there are different accelerators available, such as GPUs, FPGAs, TPUs, and DSPs, we focus on GPUs and FPGAs, since they are more general purpose than the others and can support a wide variety of applications.

GPUs have been well studied and widely used as accelerators. While GPUs are highly effective in handling applications with high level of concurrency and regular memory access patterns, they come short for applications with a high degree of dependency, and/or a high number of conditional branches. Examples of these applications include graph processing [25], sorting [26], small signal processing problems [27], and sparse linear algebra [28].

FPGAs are each a farm of logic, computation, and storage resources that can be configured dynamically. Different from widely-adopted GPUs, FPGAs can accelerate almost all types of algorithms (irrespective to their computational pattern), due to their reconfigurability. Despite their impressive acceleration power, programming and optimization difficulties have been serious obstacles to the wider adoption of FPGAs. Recent advancements in supporting high-level synthesis (HLS) have made it possible to program FPGAs using high-level languages, especially OpenCL [29], which has made FPGAs much easier to use and much more accessible to applications. Even though an HLS-based program may not perform as well as a carefully hand-crafted HDL program, the productivity enabled by HLS is often far more important.

B. OpenCL

OpenCL is a versatile C-based programming model that can execute across heterogeneous platforms, including CPUs, GPUs, and DSPs. CPU and GPU vendors, such as Intel, AMD, and NVIDIA have been supporting OpenCL on their platforms for over a decade. The recently-extended support of OpenCL to FPGAs has opened the gate for conveniently integrating FPGAs into a heterogeneous computing system. Using OpenCL, programmers do not need to make any major changes to their code, when porting it across different platforms. Moreover, developers can split their applications and deploy the parts on different accelerators to make optimal use of the accelerators' different capabilities.

OpenCL's ease of programming and portability across platforms unlock a whole new level of productivity, even though it might lose some performance compared to the traditional frameworks for accelerator programming. Compared to CUDA-based GPU programming, related works have shown OpenCL has only a slightly worse performance on GPUs [30], [31]. Compared to C-based HLS on FPGAs, OpenCL-based FPGA programming has about the same level of performance. For example, consider the disparity map calculation algorithm [32]. For the window size of 7×7 the OpenCL implementation is faster than the C one by 6.14%; For the window size of 9×9, the OpenCL implementation can be slower by up to 15.3%.

Therefore, in this study, we focus on the OpenCL-based GPU and FPGA computing and study their effectiveness for accelerating common algorithmic patterns.

C. Loop Parallelism

Algorithms are composed of one or many loops, either nested or flattened. The acceleration of algorithms is the process of accelerating the loops, using parallelization and pipelining methods. Algorithms can be parallelized either *temporally* or *spatially*.

Spatial Parallelism. In *spatial* parallelism [33], processing elements (PEs) execute the same task (SIMD) or multiple different tasks (MIMD), simultaneously. Both GPU and FPGA are able to exploit spatial parallelism in algorithms. The amount of data dependency between the iterations of the loops in the algorithm can decide the level of achievable spatial parallelism on the target architecture. In another word, having less data dependency increases the opportunity of speedup on parallel architectures, such as GPUs and FPGAs. In general, GPUs are better at exploiting spatial parallelism, because FPGAs cannot adopt as many compute cores as GPUs, and FPGAs also tend to operate at a lower clock frequency, up to 2-5 times slower than GPUs.

Temporal Parallelism. In *temporal* parallelism [33], processing tasks that have a dependency on each other are mapped onto different PEs and execute in parallel in a pipeline fashion. Data processing has multiple stages, and each stage is being handled by one PE. In this multi-stage pipeline, as data is processed by the element PE_i, it is sent to the next element

PE_{i+1} and element PE_i moves on to handle new data coming from the previous stage. In the cases where a single task cannot fully occupy the available PEs, multiple tasks can be interleaved and mapped onto the PEs to increase the *temporal* parallelism.

Among general purpose accelerators, FPGAs are exclusively able to exploit coarse-grained temporal parallelism in the algorithms, due to their reconfigurability. SIMD platforms like GPU can perform at most one instruction at a time on each available core, whereas FPGA can execute hundreds of operations on all available stages in the pipeline. Need to mention, while GPUs can launch multiple kernel streams in a pipeline fashion, they cannot achieve the fine-grained pipeline parallelism. One can mimic pipeline parallelism by launching consecutive kernels (e.g., CUDA stream kernels). Still, the data between different stages should be stored and delivered to the main memory, an expensive operation. On the contrary, FPGAs utilize connected registers between PEs to transfer the data.

Figure 1 depicts both parallelism dimensions. Each circle represents an individual iteration in a set of nested loop blocks. The (i, j) pair in each circle represents the ith iteration in the first dimension and the jth iteration in the second dimension. The arrow represents the dependency of one iteration on another, e.g., (1,2) depends on (1,1). Each iteration usually involves separate calculation for a specific indexed item or accumulation on a shared value among iterations of a loop block. The dashed box contains iterations with zero dependency, which can be easily parallelized spatially. On the other hand, the dotted box contains iterations with data dependency, which cannot be parallelizaed spatially but may have the potential to be parallelized temporally. We use the above format throughout the paper to represent the dependency flow.

In summary, GPUs excel at exploiting spatial parallelism but cannot utilize temporal parallelism, whereas FPGAs can take good advantage of both types of parallelism. However, despite this general understanding of GPU's and FPGA's different strengths, it is still difficult to understand which accelerator works the best for which algorithm. Every single application consists of different types and degrees of conditional and data dependencies. Developers usually need to implement the code for different accelerators and then apply several different transformations on the algorithm to assess the acceleration potentials on different devices. Understanding the relationship between common micro-level patterns such as loop patterns and their potential acceleration can reduce the effort of choosing the right device. These are the motivations for our study on loop acceleration using GPUs and FPGAs, which, to the best of our knowledge, is the first.

D. Automatic Loop Optimization

Automatic loop optimization (generally parallelization) dates back to an article by Lamport [24] which discusses parallel execution of *do loops*. Later, several other researchers continued the effort and developed the groundbreaking approach of using linear algebraic methods to analyze, transform,

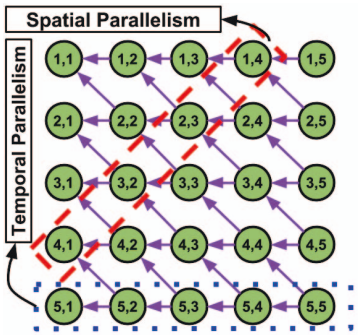

Fig. 1: Spatial and temporal parallelism in multiple iteration dimensions.

Loop Pattern	Sample Algorithm/Application
Intra-Dimension Dependency	Linear Algebraic Routines
Diagonal Dependency	Needleman-Wunsch
Conditional Dependency	Kmeans, Single-Source Shortest Path
Anti-Dependency	Floyd-Warshall Algorithm
Half-Parallelism Half-Dependency	K-Nearest Neighbor

TABLE I: List of loop blocks

and parallelize loops, namely polyhedral compilation [19]–[23].

Polyhedral compilation is used in a wide range of applications, including automatic parallelization, SIMDization, code generation for hardware accelerators, and memory and cache consumption optimization. It models nested loops and arrays into an algebraic format while presenting specific constraints, such as dependencies. Further, it uses particular types of algebraic transformation that guarantee the loop's semantic and correctness and generates a new model, typically optimized toward a specific cost model. Finally, the model is translated back into an execution code that can run on hardware.

The polyhedral compilation has limitations. First, it does not provide cross-accelerator comparisons. Such limitation prevents developers from understanding the correlation between the loop patterns and the speedup capabilities of accelerators. Second, polyhedral compilers, such as Polly [17], [18], Graphite [16], and a more recent compiler called Tiramisu [14] can only optimize specific routines in domain-specific applications, such as dense linear algebra, tensor operations, and stencil computations. Also, they are able to provide 10% performance improvement [15]. Finally, the polyhedral compilation is not well-studied on GPUs and FPGAs [11]–[13], making it less effective for accelerators.

In summary, polyhedral compilation lacks the ability to demonstrate the effectiveness of different hardware accelerators while considering an algorithm or an application. Loopy aims to unlock insights into accelerating typical loop patterns that can be generally found in many applications with important accelerators such as GPUs and FPGAs.

III. Loop Analysis

A. Methodology

Our approach to understanding how to choose the optimal accelerator for a given algorithm is by studying the perfor-

mance characteristics of common loop patterns on GPUs and FPGAs. Following this approach, we designed *Loopy*, a set of abstract and configurable loop blocks, which captures the key loop patterns extracted from real-world algorithms (Table I), and allows flexible testing of each type of loops by varying the following key parameters:

1) *Computational intensity*, which is the total number of computational operations that each iteration of the algorithm performs. In our study, it is defined as the number of multiply-accumulation operations. The computational intensity can affect the size of the pipeline and the number of instructions on both FPGA and GPU. Changing this parameter can show how both platform's performance is susceptible to the amount of computation;
2) *Dependency and concurrency degrees*, which define how many iterations depend on each other and how many other iterations can be executed separately.
3) *Input data size*, which specifies the total number of floating-point variables that the algorithm processes. The size of the input data can affect the load of computation on a target platform, which can decide the suitability of one device over another.
4) *Variable precision*, which is the bit-width size of the variables in the algorithm. FPGAs and most-recent GPUs have the capability to deliver higher performance for lower bit-precision operations.

Loopy includes optimized implementations of each loop type for GPU and FPGA. The rest of this section details each loop type and its GPU and FPGA implementations, and presents experiments from running them on real devices. While optimizing GPU programming has been well studied, *OpenCL-based FPGA optimization is not well explored and not trivial.* In our discussions, we will also detail how we performed the optimizations for each key loop type.

All GPU-related experiments were conducted on two server nodes with two type of GPUs. One server is equipped with an Nvidia Geforce RTX2080 GPU, dual Intel Xeon E5-2637 v4 CPU, and 64GB of DDR4 main memory (2133MHz). The RTX2080 is a large form-factor GPU, suitable for heavy AI and deep learning workloads. Another server is equipped with an Nvidia Tesla T4, Intel Xeon E5-2650 v3 CPU, and 198GB of main memory. The T4 is a small GPU, suitable for edge servers. All the FPGA-related experiments were conducted on an Intel Fog Reference Design unit, equipped with two Nallatech 385A FPGA Acceleration Cards (Intel Arria 10 GX1150 FPGA), and Intel Xeon E5-1275 v5 CPU, and 32GB of DDR4 main memory (2133 MHz).

The OpenCL kernels for FPGAs were compiled using Intel FPGA SDK for OpenCL (version 19.1) with Nallatech p385a_sch_ax115 board support packages (BSP). The GPU OpenCL kernels were compiled just-in-time at runtime using available OpenCL library in CUDA Toolkit 11.0. For the FPGAs, we implemented all the kernels in the single-thread mode and NDRange (multi-threaded) mode. Single-thread kernels on FPGAs typically have much less overhead and can achieve much higher clock frequency rate, compared to multi-threaded kernels. Thus we focus on the results from the single-thread mode execution on the FPGAs. For the GPUs, we implemented the kernels in the NDRange mode in OpenCL, which deploy concurrent threads on the available compute units.

The insights from our study can be generalized, irrespective of our FPGA or GPU choices. Our experiments target the general capability of accelerators in handling common loop patterns, rather than handling specific computational blocks (e.g., tensor processing for deep learning applications). Various generations of FPGAs and GPUs usually differ in their total available resources, such as programmable blocks on FPGAs and processing units in GPUs, which do not affect their general behaviors. When comparing the acceleration achieved by GPU vs. FPGA, we also focus on the general trend, i.e., how the performance changes w.r.t. the key parameters identified above, rather than the absolute performance for a specific configuration. Therefore, our characterization of loop acceleration is generally applicable to GPUs and FPGAs regardless of hardware's specific choices.

B. Intra-Dimension Dependency

Definition. This type of loops is usually composed of two or more nested iterative blocks, where each level of iterative blocks is considered a *dimension*. In this pattern there exists a *loop-carried data dependency*, which is a dependency of one iteration on the output of the previous iterations (read-after-write), in one or more dimensions, while at the same time one or more dimensions have no dependency between their iterations. In another word, we can observe both dependency and concurrency in the overall iteration space.

For example, in Algorithm 1, the dependency exists between iterations with the index of i. In this algorithm, updating every element of the array A with the index of i on the first dimension depends on the value of the element with the index of $i-1$. Elements in the second dimension with the index of j do not carry any dependency. In this case, the dependency exists on the dimension with the index of i and the concurrency exists on the dimension with the index of j. Figure 2 illustrates the iteration space and the dependency graph of intra-dimension dependent loops. Although in this example, the nested loops have only two dimensions, indexed by i and j, in reality, the algorithm can have multiple dimensions and dependency within any one of the dimensions.

Simple linear algebraic algorithms [34], such as matrix-matrix (see Listing 1) and matrix-vector multiplications are following this type of loop pattern. For example, in matrix-matrix multiplication, each cell of the output matrix can be computed separately (concurrency), while the dot multiplication of one row and one column can only be performed sequentially in a single thread (dependency).

```
1 type Row = List[Double]
2 type Matrix = List[Row]
3
```

Algorithm 1 Intra-dimension dependency algorithm

$i \leftarrow 1$
$j \leftarrow 1$
for $i \leq n$ **do**
  for $j \leq m$ **do**
    // In our case, func is an FMA operation
    $A[i][j] = func(A[i-1][j], B[i][j], ...)$
  end for
end for

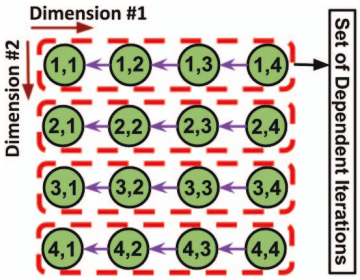

Fig. 2: Intra-dimension dependent loop pattern.

```
def dotProd(v1:Row,v2:Row) =
v1.zip( v2 ).
    map{ t:(Double,Double) => t._1 * t._2 }.
    // Dependent accumulation (Spatial Parallelism)
    reduceLeft(_ + _)

def mXm( m1:Matrix, m2:Matrix ) =
    // Parallel row-by-row multiplication (Temporal
    Parallelism)
    for( m1row <- m1 ) yield
    for( m2col <- transpose(m2) ) yield
    dotProd( m1row, m2col )
```

Listing 1: Matrix-matrix multiplication algorithm

The degree of spatial and temporal parallelism, combined with the arithmetic intensity, can determine the choice of deployment on either FPGA or GPU. Algorithms with a high degree of dependency can usually finish faster on FPGAs, while algorithms with a high degree of concurrency can utilize the available farm of SIMD compute units on the GPUs and accelerate their execution.

Implementation. Our benchmark contains the GPU and FPGA versions of the intra-dimension dependent loop. For the GPU version, the loop is unrolled spatially over the non-dependent dimension. Each independent iteration is deployed as a work-item (unit of a task in OpenCL), and all the work-items are grouped into several work-groups (unit of execution on a single compute unit). Also, we specifically order the memory access indexes to enable memory access coalescing among work-items in a work-group for better performance. For the FPGA version, we first apply statement re-ordering to place the dependent loop as the inner-most loop, which enables interleaving of the outer-loop iterations (non-dependent) inside the inner-loop cycle. It also helps achieve the initiation interval of one in the inner-most loop. In loop pipelining, the initiation interval is the number of clock cycles between the start times of consecutive loop iterations. Having an initiation interval of one enables the FPGA to push one iteration into the pipeline at every clock cycle and achieve the highest performance, which is the ultimate goal for every design. Further, we apply *loop blocking* (also known as loop tiling) on the outer for loop. Doing so enables utilization of the on-chip registers on the FPGA (with the same size of the block), by copying the required data for the execution of the block, as a whole, onto the allocated on-chip registers, thereby reducing the DRAM access overhead.

Experiment. We deployed FPGA and GPU kernels, resembling Algorithm 1. Input data is an array of floating-point variables of a specific size (4, 32, 256 MB). Every single iteration in the algorithm is responsible for a single element in the array. As a result, the total number of iterations is equal to the number of input values. As shown in the algorithm, the dependency and concurrency degrees are configured by changing the number of iterations, n and m, respectively. Figure 3 shows the runtime of this intra-dimension dependent loop on both FPGA and GPU.

We can make several key observations from the results. First, GPU does excel at accelerating the loop with a high degree of concurrency. More concurrency can lead to better spatial parallelization, which makes the GPU a great candidate for deployment. In contrast, with the increase in the dependency degree, the FPGA can take advantage of the configured long pipeline and parallelize the dependent iterations. In this case, with a high degree of dependency, the FPGA can outperform both RTX2080 and T4 by up to 21.5x and 13.6x. With a high degree of concurrency, both RTX2080 and T4 perform better than the FPGA, by up to 184x and 93x, respectively.

The second observation is about the effect of computational intensity (the total number of computational operations in each iteration) on the final performance. Higher intensity means more computations, which leads to more pipeline stages. With more pipeline stages, FPGA can handle more dependent iterations and achieve higher performance. Need to mention, the available hardware resources on the FPGA are limited and may block developers from configuring a large number of pipeline stages. As a result, developers may need to adopt a smaller loop block size, which leads to the reduction of performance. Compared to FPGA, the GPU has to spend more time executing each loop iteration, with no opportunity for pipelining the iteration. For example, Figure 3 shows that going from the intensity of 1 to 5, the performance drops by up to 2.1x and 3x, on RTX2080 and T4, respectively.

The third observation is the performance reduction of FPGA for kernels with low dependency, because there are not enough dependent iterations to fully saturate the configured pipeline. In this situation, developers may want to switch to the NDRange mode kernels, which can interleave the parallel iterations into the pipeline and keep it saturated. In comparison, GPU can utilize the massive farm of cores to exploit a high degree of parallelism when the dependency is low. Therefore, as shown in Figure 3, FPGA's performance is worse with lower dependency degree whereas GPU's performance is not affected.

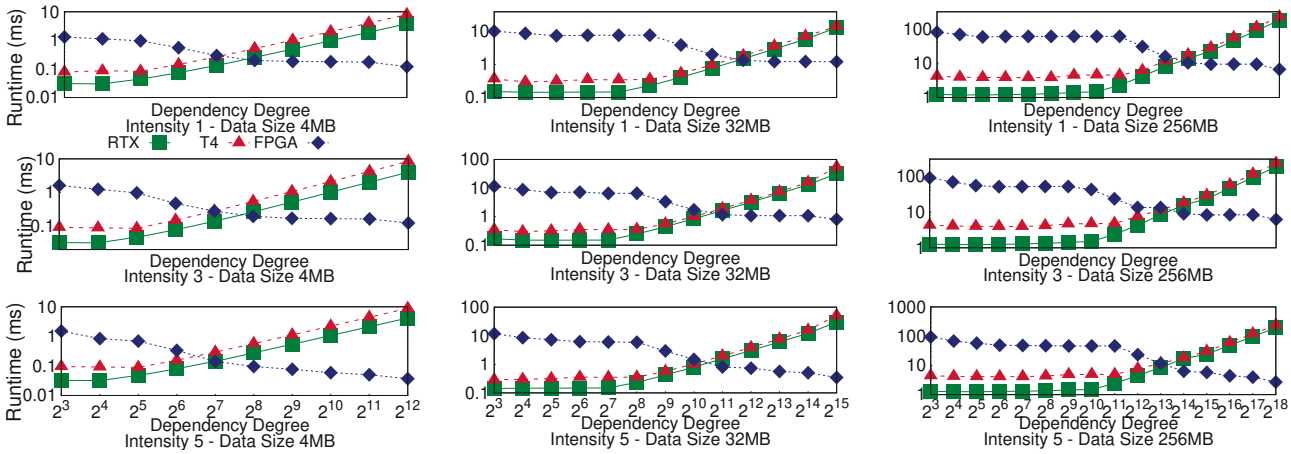

Fig. 3: Intra-dimension dependency performance on the GPU and the FPGA

C. Diagonal Dependency

Definition. Diagonal dependent loops are following almost the same pattern as intra-dimensions dependent loops, except that the dependency is diagonal instead of horizontal or vertical in the iteration space. As illustrated in Figure 4, horizontal (vertical) dependency refers to the dependency of an iteration on the left (top) neighbor iterations with the same i (j), respectively. For example, in the aforementioned intra-dimension dependency, there is horizontal dependency among the iterations as shown in Figure 2. Diagonal dependency means that an iteration depends on its relative top-left iteration which has both different i and j indexes. For example, in Figure 4, iteration $(2, 2)$ depends on its diagonal neighbor iteration $(1, 1)$. Algorithm 2 shows an example of this kind of loops, where the computation requires data from its diagonal neighbor in the iteration space. In specific cases, the dependency can be extended and include either horizontal or vertical, as well.

Parallelization of these types of loops on SIMD architectures, such as GPU, is not straightforward. Depending on the type of diagonal dependency, developers can either parallelize the diagonals or use the wavefront technique [35] for parallelization. In the wavefront parallelism mode, kernels are enqueued back to back to the GPU, each computing one set of independent iterations. The number of the kernels is equal to the length of the diagonal.

Dynamic programming algorithms are usually composed of diagonal dependent iterations. A specific example of such algorithm is Needleman-Wunsch [36] (see Listing 2), which performs matching between two input strings while minimizing the penalty.

Algorithm 2 Diagonal depedency algorithm

```
i ← 1
j ← 1
for i ≤ n do
    for j ≤ m do
        A[i][j] = func(A[i − 1][j − 1], B[i][j], ...)
    end for
end for
```

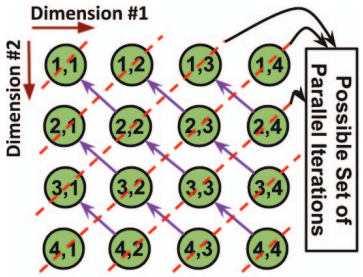

Fig. 4: Diagonal dependency loop pattern

```
val alignMat = new Array[Array[ResultEntry]](length)
val contant = ...
val gapPenalty = ...
def getScore(i: Int, j: Int): Score = {
    if (alignMat(i)(j) != null) {
        alignMat(i)(j)
    } else {
        // 3-Way Diagonal Dependency
        val tryMatch = getScore(i - 1, j - 1) +
                       constant
        val horizontalGap = getScore(i, j - 1) +
                       gapPenalty
        val verticalGap = getScore(i - 1, j) +
                       gapPenalty
        if (m == tryMatch) {
            alignMat(i)(j) = (m)
        } else if (m == horizontalGap) {
            alignMat(i)(j) = (m)
        } else {
            alignMat(i)(j) = (m)
        }
        m
    }
}
```

Listing 2: Needleman-Wunsch algorithm score calculation

Implementation. For the GPU implementation, the parallelization method depends on the existence of vertical or horizontal dependency. In the absence of both of these dependencies, each thread can take care of one diagonal, in parallel. The existence of any of the mentioned dependencies (in addition to diagonal dependency) would force the GPU to perform *anti-diagonal parallelization*. As shown in Figure 4,

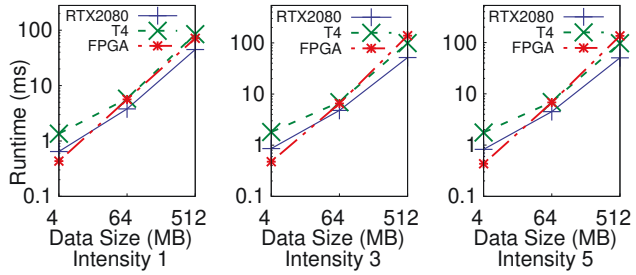

Fig. 5: Diagonal dependency runtime on both FPGA and GPU. The dependency is only diagonal.

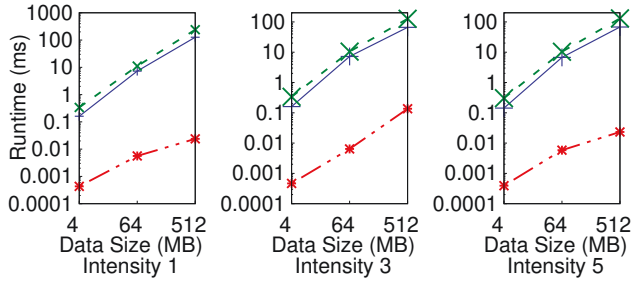

Fig. 6: Diagonal dependency runtime on both FPGA and GPU. The dependency also includes horizontal and vertical.

the independent iterations that can be parallelized form a line that is perpendicular to the diagonal dependent iterations.

For the FPGA implementation, we first perform loop blocking on the first dimension, which enables caching of the input data for each iteration of the second dimension's iterations. Later, we copy the required data for the second dimension's computation into the allocated on-chip registers of the size block. Every iteration of the second dimension first reads the data from the registers, performs the calculation, and writes back the data to the registers and the DRAM. To handle all elements in the block, each iteration of the second dimension contains a nested loop of size block, which is fully unrolled. In this implementation, the iterations of the second dimension have a loop-carried data dependency. Unfortunately, the compiler cannot infer an initiation interval of one for this loop body, due to the existence of large latency between consecutive iterations of the loop. To overcome this issue, we interleave the execution of the block iterations inside the second dimension loop, which enables full exploitation of the available pipeline stages. Doing so reduces memory accesses and leads to higher operating frequency and fewer stalls in the pipeline.

Experiment. Figure 5 shows the performance of the diagonal dependent loops on the FPGA and the GPUs, where the dependency only exists diagonally. We did measurements for three different computational intensities (1, 3, and 5) and three different input sizes (4, 64, and 512 MBs). The results show that the GPU outperforms the FPGA in almost all cases, except for the experiment with high computational intensity and small data size. In this type of dependency, GPU can assign one diagonal set of iterations to one work-item and exploit high degree of parallelism on all the available cores. In this case, RTX2080 and T4 outperform the FPGA by up to 6x and 4.3x, respectively.

Figure 6 shows the performance of the same loop pattern but with additional horizontal and vertical dependencies between the iterations. We modified the function f in Algorithm 2 to include both $A[i-1][j]$ and $A[i][j-1]$, in addition to $A[i-1][j-1]$, as its parameters to introduce these dependencies between $A[i][j]$ and its horizontal, vertical, and diagonal neighbor iterations. In this case, the FPGA can utilize the same pipelining method to accelerate the execution, while both GPUs need to use wavefront parallelism to parallelize computation for each anti-diagonal. Unlike the case with diagonal dependency, the wavefront parallelism model cannot exploit a large number of parallel threads. In addition, it needs to repetitively deploy the same kernel to calculate a new set of anti-diagonal iterations. As a result, the FPGA outperforms both RTX2080 and T4 by up to 165x and 322x, respectively.

D. Conditional Dependency

Definition. The existence of conditional statements in loop bodies can alter the extent of parallelization on certain accelerators. In loops with a conditional statement, every iteration diverges in the execution path, depending on the specific conditions. Algorithm 3 represents an example, where every iteration performs either the first or the second statement based on the content of an array in that specific iteration index.

Algorithms such as K-means and single-source shortest path (SSSP) consist of many conditional decisions. In the K-means (see Listing 3), the clustering of the observations requires many comparisons, based on the distance; SSSP relies on the sparse matrix multiplication, where the number of iterations for each output calculation is non-deterministic.

Algorithm 3 Conditional dependency algorithm

$i \leftarrow 1$
for $i \leq n$ **do**
  if $B[i] > 0.0f$ **then**
    $A[i] = f(B[i], D[i], ...)$
  else
    $A[i] = f(C[i], D[i], ...)$
  end if
end for

```
def run(xs: List[Point]) = {
    var centroids = xs take n

    for (i <- 1 to iters) {
        centroids = clusters(xs, centroids) map
    average
    }
    clusters(xs, centroids)
}

def clusters(xs: List[Point], centroids: List[Point
    ]) =
    (xs groupBy { x => closest(x, centroids) }).
    values.toList

def closest(x: Point, choices: List[Point]) =
    // Calculating mininum, which requires several
    comparisons (if-else)
    choices minBy { y => dist(x, y) }
```

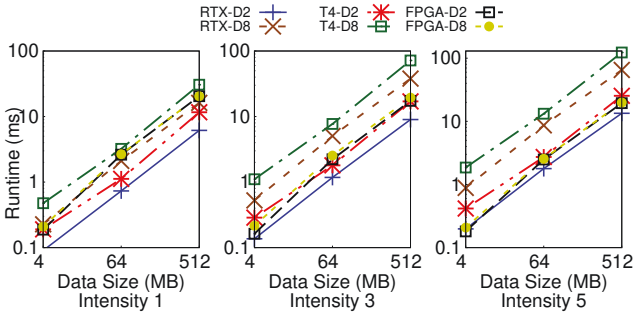

Fig. 7: Conditional dependency runtime on both FPGA and GPU, for different intensities.

```
16
17 def dist(x: Point, y: Point) = (x - y).modulus
18 def average(xs: List[Point]) = xs.reduce(_ + _) / xs
       .size
```

Listing 3: Kmeans algorithm

Implementation. The conditional dependency is introduced by an if-else statement in the kernel. On the GPU, the loop is simply parallelized on different cores, and each thread performs the if-else comparisons. But the SIMD architecture in the GPU cannot efficiently handle the conditional statements in the work-items, due to thread divergence issue. In the FPGA implementation, the kernel is developed in a single-thread mode and the loop is unrolled to the limit of the FPGA area and available DRAM bandwidth. In contrast to the GPU implementation, FPGAs can handle numerical conditional statements, using look-up tables and a simple multiplexer. More specifically, the FPGA can map all different paths of the execution in the design and enable different threads running simultaneously in different conditional blocks.

Experiment. Figure 7 shows the runtime of the conditional dependent loop on the GPU and FPGA, with various computational intensities (one, three, and five) number of conditional branches (two and eight) within each iteration as well as various total input data sizes (4, 64, and 512 MB). The number of conditional statements is represented as D2 and D8, for two and eight conditional decisions, respectively. The results show that the FPGA can sustain the same performance among kernels with different conditional branches, whereas the GPU suffers more performance degradation for kernels with more conditional branches (up to 45% slowdown). As a result, the FPGA outperforms the GPU with a higher number of conditional dependencies; e.g., 40% better for a dependency level of eight. This observation suggests the suitability of FPGAs for algorithms with a high degree of decision making during the execution. These types of applications usually cannot exploit the massive parallelism in SIMD architectures and can be better handled by reconfigurable processors.

E. Anti-dependency

Definition. In this loop pattern, every iteration consists of more than one statement. Unlike the intra-dimension dependent loops, where the dependency is read-after-write, this pattern carries write-after-read dependency. In this pattern, one statement of an iteration reads a data item that is going to be updated by the other statement in the next iteration. It is named anti-dependency because the statements in different iterations are following the write-after-read pattern, as opposed to read-after-write in the typical dependency patterns. Algorithm 4 demonstrates a general example of such loops. The existence of read-after-write dependency creates an anti-dependent loop pattern.

Anti-dependent loops have a unique characteristic. It is possible to face race condition in case of parallelization of all the iterations. More specifically, the first iteration reads the old value of an array element (e.g., $A[i]$ depends on $B[i+1]$ in Algorithm 4), while the second iteration updates the same value, and so on and so forth. When these iterations are executed on different threads to achieve parallelism, the dependent read and write might be executed out of order, which damages the correctness.

Algorithm 4 Anti dependency algorithm

$i \leftarrow 1$
for $i \leq n$ **do**
  $A[i] = B[i+1] + C[i] * D[i]$
  $B[i] = B[i+1] - E[i] * D[i]$
end for

Implementation. These types of loops can be parallelized on vector processors with a global barrier mechanism among all SIMD threads. Unfortunately, both the FPGA and the GPU lack such a global barrier mechanism between all threads. An approach to parallelizing inter-iteration dependent loops is loop-splitting. In this approach, the loop can be divided into multiple separate loops, where none of them carries any dependency. In this situation, loops should run sequentially on the target processor (to guarantee the correctness of the execution), but each loop can fully exploit the available spatial core units. Figure 8 represents the execution and the dependency of the original loop, along with the transformed version of it. The dotted blue box and the solid red box represent different statements in the loop body. The arrow shows the anti-dependency between different statements of consecutive iterations.

To accelerate anti-dependency loops on GPU and FPGA, we apply statement re-ordering and loop splitting. The transformation creates multiple flattened loops, where each of them represents a stage of the execution. The lack of global barriers prevents both platforms from co-locating the execution of the generated sub-loops after the main loop distribution, except for using channels in FPGA, which is a mechanism for passing data between kernels and synchronizing kernels with high efficiency and low latency. Usually, kernels need to communicate through DRAM, which increases the application runtime. By using channels, loops can start pipelining their partial results to the next loop, which enables co-location of the computation and communication and reduces the application runtime. Unlike the FPGA, GPU should execute the flattened loop sequentially, but each stage can be fully parallelized spatially.

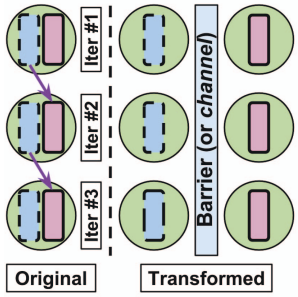

Fig. 8: Anti dependency loop pattern.

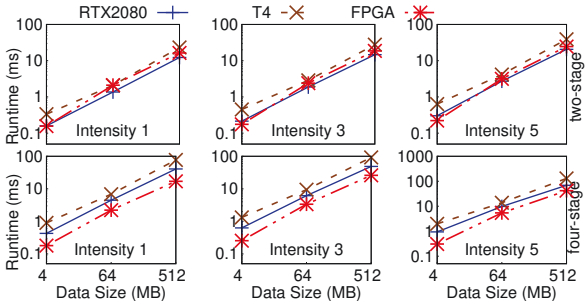

Fig. 9: Anti dependency results for two and four stages.

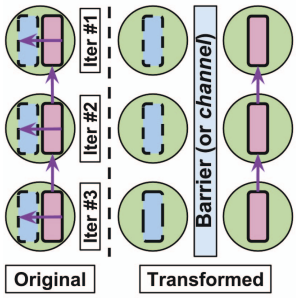

Fig. 10: Half-parallellism half-dependency loop pattern.

Experiment. Figure 9 shows the runtime of the FPGA and GPU in accelerating these loops. We varied the degree of anti-dependency which is the number of statements involved in the anti-dependency. For example, in Algorithm 4 the dependency exists between two statements, which yields anti-dependency degree of two. As a result, the main loop in the benchmark can be split into several separate and parallelizable loops, depending on the number of anti-dependent statements in the loop body. We also varied the intensity level and input data size.

Comparing the runtimes for the case of four stages of anti-dependencies, the FPGA can outperform the Tesla K40 GPU for kernels with low intensity (up to 20% speedup), whereas it performs close to the GPU for higher intensities (up to 15% speed degradation). Comparing to Titan X, the FPGA performs 1.6x slower. Kernels with higher intensities lead to larger area consumption and limit the parallelism level in each stage, which results in the reduction of the channels widths. Overall, increasing the number of statements with anti-dependencies results in more separate loops. As shown in Figure 9, increasing the degree of anti-dependency reduces the gap between the FPGA and GPU. We can expect that by following this trend, the FPGA will eventually outperform the GPU.

F. Half-Parallelism Half-Dependency

Definition. Half-parallel half-dependent loops usually include the dependent and the parallel statements, simultaneously, and consist of only one loop, with no nested loop. Algorithm 5 lists an example of this type of loops. The existence of loop-carried dependent statements (read-after-write) prevents the spatial parallelization of the algorithm, as a whole. Transforming the loop into multiple flattened loops enables the execution of the loop in two different stages. Unlike the anti-dependent loops, the loop-splitting process does not enable spatial parallelization opportunity for all the loops, since part of the algorithm carries read-after-write dependency. After the splitting, the parallel portion of the loop can be deployed on processors with a high number of parallel compute units, e.g., GPUs, while the dependent portion can be handled by processors that are suitable for sequential execution, e.g., CPUs and FPGAs.

Figure 10 represents the half-parallelism half-dependent loop pattern. For this pattern, each red box in an iteration depends on another red box from the previous iteration. Furthermore, each red box depends on the value of the blue box in the same iteration.

Half-parallel half-dependent applications such as K-nearest neighbor (KNN) include of both parallel parts (distance computation) and dependent parts (sorting) (see Listing 4). These applications can utilize one or more hardware accelerators for an efficient acceleration.

```
val sortedDistances = data.map{case (a, b)
        => (b, Util.euclideanDistance(p, a))}
        .sortBy(_._2, ascending = true)
// take the top k results
val topk = sortedDistances.zipWithIndex()
        .filter(_._2 < k)
// take the most predominant class within the top k
val result = topk.map(_._1)
        // Parallel section of the KNN
        .map(entry => (entry._1, 1))
        .reduceByKey(_+_)
        // Semi-Dependent section of the KNN
        .sortBy(_._2, ascending = false).first()
```

Listing 4: KNN algorithm

Implementation. We apply loop splitting to separate the parallel section from the dependent section. For the GPU, we first compute the parallel part on the GPU and then transfer the data back to the main memory of the host and execute the dependent part on the CPU. Running the dependent block of code on the GPU is not efficient and will lead to poor performance. For the FPGA we have multiple options, (1) running the parallel and dependent blocks of the loop serially on the FPGA, (2) running the parallel block on the FPGA and the dependent block on the CPU, and (3) using channel to pipeline the intermediate result from the parallel part to the dependent part and decrease the running time overhead. Using the channels is the best available option to co-locate computation and communication and achieve the highest possible performance.

Experiment. Figure 11 shows the runtime of the FPGA and GPUs in acceleration these loops. For this experiment, we provided input data with a size of 1 to 1024 MB. The FPGA can outperform both Titan X and Tesla K40 GPUs, by up to

Algorithm 5 Half-parallelim half-dependency algorithm

$i \leftarrow 1$
for $i \leq n$ **do**
  $A[i] += C[i] * D[i]$
  $sum += B[i] + A[i] + D[i]$
end for

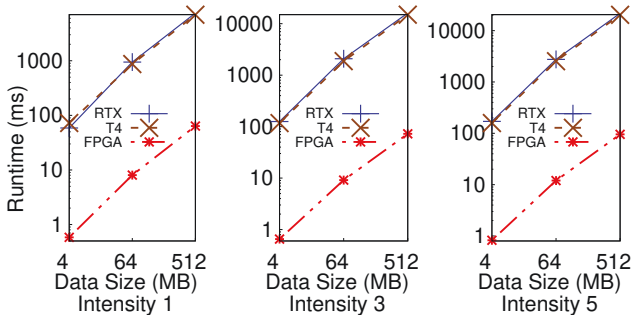

Fig. 11: Half-parallelism half-dependency runtime on both FPGA and GPU, for different intensities.

118x and 110x, respectively. The overhead of the data transfer from the GPU to CPU reduces both GPUs' performance significantly. As a conclusion, co-locating the parallel and the dependent sections of the code on the FPGA can yield much higher performance, compared to utilizing GPU+CPU combination with a much slower communication channel.

IV. Related Work

To the best of our knowledge, we are the first to provide a comprehensive study of common loop patterns on important hardware accelerators, including both GPUs and FPGAs. There are a number of related works that are complementary to the focus of our study. Roofline modeling [37] was first designed to provide insights into the performance of multicore architectures, utilizing a parameter, operational intensity. It helps understand the potential bottlenecks and improvement opportunities for an application on different families of CPUs. Other efforts [38], [39] extended this model to accelerators, such as the GPU and TPU. The roofline model does not provide insights into the loop-level acceleration opportunity on different hardware accelerators. In comparison, Loopy provides optimization details in loop-level granularity (not the whole application) and does not rely on the real implementation of the algorithm.

Existing benchmarks adopted widely-used algorithms or computational patterns to draw comparison lines between different processors. Some of these works [40], [41] focused only on a particular type of processor, whereas others [42] were designed to compare different families of processors, e.g., CPUs vs. GPUs. These benchmarks help understand the performance differences between accelerators while executing certain types of applications, but their insights are limited to specific applications. It is difficult for a developer to use these benchmarks to decide which accelerator has more potential to accelerate a new type of application. In comparison, Loopy offers insights into accelerating common loop patterns, which are not limited to a particular application and can be applied to any new algorithm.

Closely related to our approach, the TSVC benchmark [43] includes a suite of various types of loops, which has inspired some of the loop patterns considered by our Loopy. TSVC was mainly designed to evaluate the efficiency of compilers on detecting and vectorizing such loops on SIMD architectures. In comparison, the goal of Loopy is to evaluate the correlation between common loop patterns and the extent of accelerating such loops on different hardware platforms. In addition, it provides an in-depth analysis of how loop characteristics impact the accelerator performance, all of which are not possible by simply applying or porting TSVC.

There are efforts in predicting the performance of a complete application on a target platform [44], [45]. These solutions require access to the real implementation of the application, and the prediction is specific to the application. In comparison, Loopy is able to give insights into the acceleration opportunities at the abstraction level of loop patterns.

Finally, polyhedral compilation is a body of works aiming for transforming loops in the application to achieve a higher performance [14]–[21]. It represents loops in algebraic form as polyhedra, and further applies algebraic transformation. Each transformation optimizes a certain aspect of the code, such as parallelization. In comparison, Loopy aims to provide insights into the capability of different accelerators for loop acceleration, before any transformation.

V. Conclusions and Future Works

In this work, we designed Loopy for studying common loop patterns on important GPU and FPGA accelerators. We identified and analyzed five common loop patterns, along with the key configuration parameters in these patterns. We then studied the acceleration opportunities for these loop patterns and how the loop configurations and accelerator platforms affect the effectiveness of acceleration. Using Loopy, developers can gain a good understanding of the acceleration potential of their algorithms on different platforms, without having to implement them for any specific platform, based on the loop patterns that these algorithms embody. LoopBench is open source and publicly available[1].

Understanding the performance of applications consisting of multiple algorithmic patterns and optimizing the partitioning and placement of an application across heterogeneous processors/accelerators are the ultimate objectives of our research. Loopy provides an important first step towards the optimized use of accelerators for diverse applications in heterogeneous computing systems.

VI. Acknowledgment

We thank the anonymous reviewers for their helpful comments. This work is supported by National Science Foundation awards CNS-1955593, CNS-1562837, and CNS-1629888 and Intel's donation of the Fog Reference Design units.

[1]https://github.com/saman-aghazadeh/loopy

REFERENCES

[1] J. D. Owens et al., "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.

[2] S. Biookaghazadeh, F. Ren, and M. Zhao, "Are FPGAs suitable for edge computing?" *arXiv preprint arXiv:1804.06404*, 2018.

[3] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*. ACM, 2012, pp. 47–56.

[4] Google, "Google Cloud GPU," https://cloud.google.com/gpu, 2021.

[5] ——, "Google cloud tpu," https://cloud.google.com/tpu, 2021.

[6] Microsoft, "Microsoft azure gpu," https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu, 2021.

[7] ——, "Microsoft azure fpga," https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service, 2021.

[8] Amazon, "Amazon aws gpu," https://docs.aws.amazon.com/dlami, 2021.

[9] ——, "Amazon aws fpga," https://aws.amazon.com/ec2/instance-types/f1/, 2021.

[10] H. R. Zohouri et al., "Evaluating and optimizing opencl kernels for high performance computing with fpgas," in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*.

[11] J. Wang, L. Guo, and J. Cong, "Autosa: A polyhedral compiler for high-performance systolic arrays on fpga," in *Proceedings of the 2021 ACM/SIGDA international symposium on Field-programmable gate arrays*, 2021.

[12] J. C. Juega, J. I. Gómez, C. Tenllado, and F. Catthoor, "Adaptive mapping and parameter selection scheme to improve automatic code generation for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 251–261.

[13] A. Konstantinidis, P. H. Kelly, J. Ramanujam, and P. Sadayappan, "Parametric gpu code generation for affine loop programs," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2013, pp. 136–151.

[14] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2019, pp. 193–205.

[15] A. Simbürger, S. Apel, A. Größlinger, and C. Lengauer, "The potential of polyhedral optimization: An empirical study," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 508–518.

[16] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta, "Graphite two years after: First lessons learned from real-world polyhedral compilation," in *GCC Research Opportunities Workshop (GROW'10)*, 2010.

[17] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[18] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[19] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.* IEEE, 2004, pp. 7–16.

[20] V. Loechner, "Polylib: A library for manipulating parameterized polyhedra," 1999.

[21] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1991, pp. 39–50.

[22] R. Schreiber, J. J. Dongarra *et al.*, *Automatic blocking of nested loops*. Research Institute for Advanced Computer Science, NASA Ames Research Center, 1990.

[23] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1978, pp. 84–96.

[24] L. Lamport, "The parallel execution of do loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, 1974.

[25] J. Cong et al., "Understanding performance differences of FPGAs and GPUs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

[26] D. Koch and J. Torresen, "FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 45–54.

[27] B. Duan et al., "Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU," in *Field-Programmable Technology (FPT), 2011 International Conference on*. IEEE.

[28] Y. Zhang et al., "FPGA vs. GPU for sparse matrix vector multiply," in *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pp. 255–262.

[29] A. Munshi, "The opencl specification," in *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE, 2009, pp. 1–314.

[30] C. Nugteren, "Clblast: A tuned opencl BLAS library," *arXiv preprint arXiv:1705.05249*, 2017.

[31] J. e. a. Cong, "Best-effort FPGA programming: A few steps can go a long way," *arXiv preprint arXiv:1807.01340*, 2018.

[32] S. Qin and M. Berekovic, "A comparison of high-level design tools for soc-fpga on disparity map calculation example," *arXiv preprint arXiv:1509.00036*, 2015.

[33] A. A. Freitas and S. H. Lavington, "Basic concepts on parallel processing," in *Mining Very Large Databases with Parallel Processing*. Springer, 2000, pp. 61–69.

[34] G. Guennebaud, B. Jacob *et al.*, "Eigen," *URl: http://eigen. tuxfamily. org*, 2010.

[35] M. E. Belviranli et al., "Peerwave: Exploiting wavefront parallelism on gpus with peer-sm synchronization," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 25–35.

[36] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[37] Y. J. Lo et al., "Roofline model toolkit: A practical tool for architectural and program analysis," in *Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*.

[38] H. Jia, Y. Zhang, G. Long, J. Xu, S. Yan, and Y. Li, "GPURoofline: a model for guiding performance optimizations on GPUs," in *European Conference on Parallel Processing*. Springer, 2012, pp. 920–932.

[39] D. Doerfler et al., "Applying the roofline performance model to the intel xeon phi knights landing processor," in *International Conference on High Performance Computing*. Springer, 2016, pp. 339–353.

[40] S. Che, M. Boyer, and et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[41] G. Ndu et al., "CHO: towards a benchmark suite for OpenCL FPGA accelerators," in *3rd International Workshop on OpenCL*. ACM.

[42] A. e. a. Danalis, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 63–74.

[43] S. Maleki et al., Y. Gao, and M. J. Garzar, "An evaluation of vectorizing compilers," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*. IEEE, 2011, pp. 372–382.

[44] S. Kumar, V. Srinivasan, and et al., "Peruse and profit: Estimating the accelerability of loops," in *Proceedings of the 2016 International Conference on Supercomputing*. ACM, 2016, p. 21.

[45] J. Meng et al., "GROPHECY: GPU performance projection from CPU code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 14.