# EECE 5640 High Performance Computing
# Project Report

Jinghan Zhang
zhang.jingh@husky.neu.edu
NUID: 001105650

Aly Sultan
sultan.a@husky.neu.edu
NUID: 001057787

Professor: David Kaeli
TA: Elmira Karimi

April 2020

# Domain Design Space Exploration of ACC-Rich Platform for Many Streaming Applications

## 1 Motivation

Heterogeneous accelerator-rich (ACC-rich) platforms combining general-purpose cores and specialized HW accelerators (ACCs) promise high-performance and low-power streaming application deployments, e.g. for video analytics, software-defined radio, and radar. In order to recover Non-Recurring Engineering (NRE), a unified domain platform for a set of applications is desirable. When applications within one domain have functional and structural similarities, they can benefit from common ACCs [6]. One challenge is to identify the most beneficial set of ACCs to deploy them on a unified domain platform.

However, current allocation strategies mostly produce a dedicated platform for one application in isolation. Automatically exploring the allocation of a domain platform for many applications requires broadening the scope to many applications [8]. In this domain design space exploration (DSE) of ACC-rich platform, there are two challenges. First, many applications increase the number of ACCs candidates needed to be considered, and the design space of platform allocation is tremendously enlarged. Second, the number of application performance evaluation for each allocation is also increased. Due to a large number of evaluations are required, the domain DSE is tremendously slow.

## 2 Contribution

To achieve an efficient domain DSE, we will apply OpenMP/MPI to parallel design space traversal. Multiple places parallelism will be considered, e.g. in application analysis, platform allocation, binding and evaluation.

To speed up the platform performance evaluation, this project will translate the current application evaluation (implemented in cpp) into matrix processing, and run it on GPU. Also, a generator will be introduced to provide evaluation matrices to represent application information, platform allocation, as well as their binding. Different bindings will be run at the same to increase the parallel, by merge their information matrices together.

## 3 Domain DSE

In this project, the domain design space exploration (Domain DSE) problem is defined as follows. Given a set of applications and a HW budget (area), find a platform ACC allocation to maximize overall performances across all applications.

The flow of the current domain DSE solution is shown in Fig. 1. The domain DSE first analyzes all applications and obtains ACC candidates (blue part). Then, the allocation algorithm chooses ACCs to deploy them on the platform (green part). To get individual application performance on this platform, binding algorithm and evaluation are applied on each application (orange part). At last, an aggregation of all application performances gives the platform performance for all applications (green part). The rest of this section gives the detail implementation information of each part in the Domain DSE.

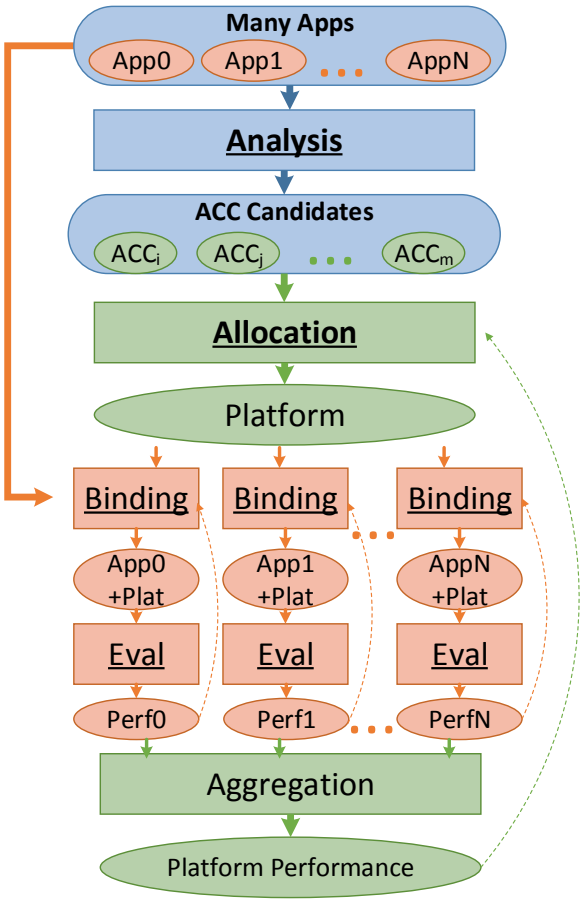



Figure 1: Domain DSE Flow

**Applications**. In this project, the streaming applications are abstracted into data flow models [4]. 40 OpenVX (computer vision) applications from Intel [3] and AMD [1] has been studied. They contains 35 different functions (38 functions if considering decomposition). The OpenVX functions are given in [2], which contains both complex functions (such as $CannyEdgeDetector$, $OpticalFlowPyramid$, and $HarrisCorner$) and simple functions (such as $Phase$, $Thresholding$, and $Accumulate$).

**Analysis and ACC Candidates**. The analysis of applications generates different ACC candidates, according to the functions. The candidates of a complex function contain both one holistic ACC and multiple small composable ACCs. For example, function $CannyEdgeDetector$ can be accelerated on both holistic $ACC_{CannyEdgeDetector}$ or composable $ACC_{SobelFilter}$, $ACC_{Magnitude}$, $ACC_{Phase}$, $ACC_{Hysterysis}$. Both accelerations achieve the same latency. Multiple small composable ACCs have higher flexibility (higher probability to be reused for other functions) than the holistic ACC. However, the composable ACCs take more area and power consumption, due to more memory is needed to store data transfer between ACCs. In the OpenVX domain, there are 37 different ACC candidates.

**Allocation**. Allocation is choosing the ACC candidates to deploy on the platform, and there are a large number of possible allocation. Just assuming the HW area budget allows 8 ACCs, then the number of possible allocation is $37^8 = 3512$ billion.

Two algorithms are applied to explore the allocation design space. One is the exhaustive search, which will explore all possible platform allocations. Another algorithm is GIDE [9], which is an advanced elitist genetic algorithm (GA).

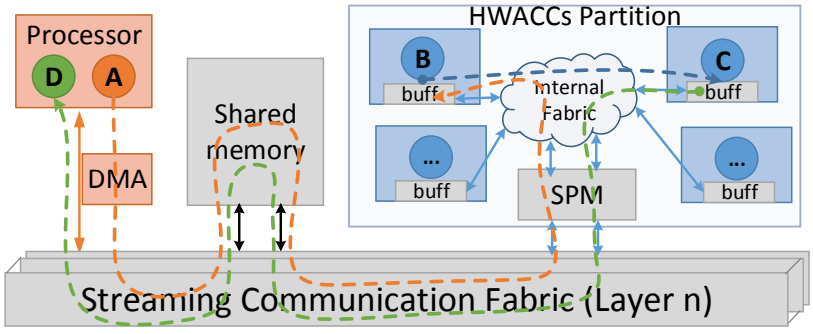



Figure 2: Target Platform

**Platform**. Fig. 2 illustrates the target platform for the context of this work. A streaming

application $A->B->C->D$ is mapped across the SW processor and HW. Each HW ACC has a dedicated Scratch Pad Memory (SPM). All ACCs are aggregated to a HW partition with a shared SPM across all ACCs. ACCs can communicate with each other, their communication traffic is hidden from system communication fabric (e.g. $B$, $C$), and the communication overlaps processing using double buffering. For simplicity, we assume direct n:n communication within the HW partition. Conversely, ACCs and SW kernels communicate through the system streaming fabric (e.g. $A$ to $B$, and $C$ to $D$) which results in throughput penalties.

**Binding**. Binding decides application functions mapped on which platform processing elements (processor or ACCs). For example, in Fig. 2, function $A$, $D$ are mapped on SW processor, and function $B$, $C$ are on two ACCs. For each pair of one application and one platform, there is a huge number of possible bindings. Assuming one application contains 10 functions, and each function could be mapped on 2 different ACCs, then the number of possible bindings is $3^{10} = 59$ thousand (base 3 is 1SW + 2ACCs).

To explore the possible bindings, two basic algorithms have been used in this project. If the number of bindings is small (less than 10 thousand), the exhaustive search is applied to achieve the optimal binding. If the number of bindings is large, multiple iterations of local search are used to find a good local optimal binding.

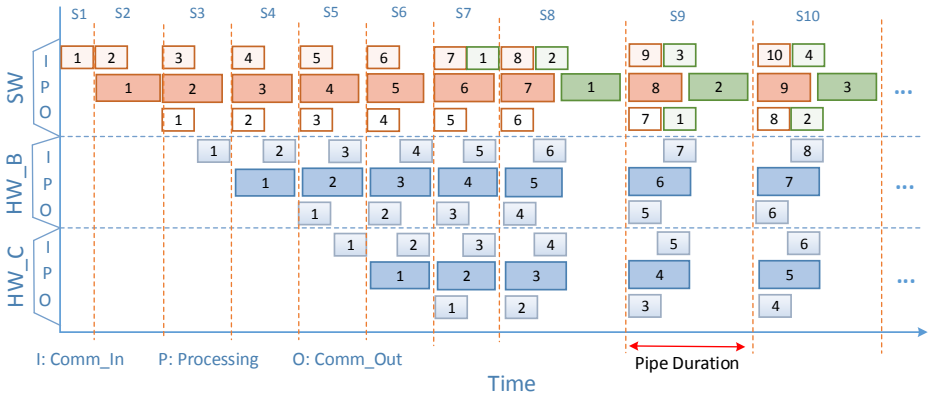



Figure 3: Pipelined Execution on Target Platform (Fig. 5)

**Evaluation**. The kernels in a streaming application operate as producers / consumers creating a pipeline. Fig. 3 visualizes the pipelined execution of a streaming application $A, B, C, D$ when mapped as shown in Fig. 2 to a platform consisting of two ACCs and one SW core. Each execution stage overlaps communication (In/Out) with processing (P) due to double buffering. Nodes execute concurrently given their dependencies across different components. Nodes mapped to the same component execute sequentially (e.g. $A, D$ in *S8*).

Our Analytic Performance Estimation (APE) model computes the throughput of an application based on the inter-kernel pipelined execution [5]. At this abstraction, the throughput only depends on the pipe duration which is determined by the slowest processing component (or communication). The analytic model uses published measurements for energy estimation (detailed references in [7]).

**Aggregation and Performance**. A fair aggregation of all application performances produces the platform performance to assess its fitness. In this project, the metric of platform performance is normalized performance, latency or energy efficiency (throughput per watt), and the details are in [8].

# 4 Project Implementation and Result

The current Domain DSE program is implemented using cpp, which is available on gitlab (`https://gitlab.com/nteimouri/PDF.git`).

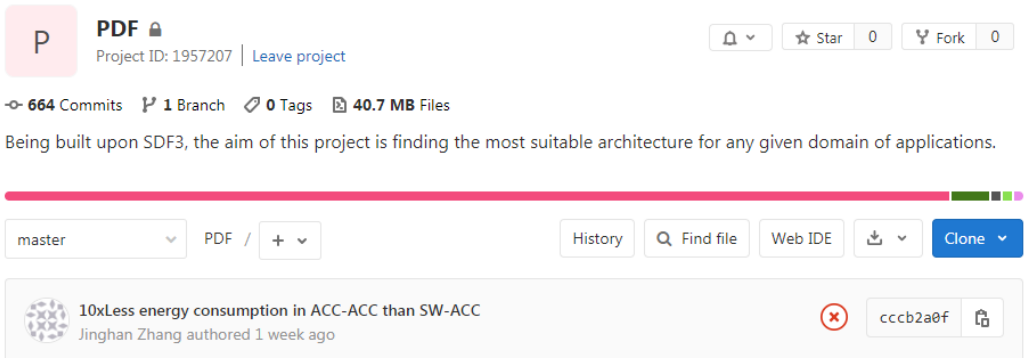



Figure 4: Project on Gitlab

As Fig. 4 shown, the total program size is 40.7 MB. The program execution is sequential and slow. Assuming 0.6 ms for individual application evaluation [9], the program will take around $4.97 * 10^6$ billion seconds to find optimal OpenVX platform allocation using exhaustive search (3512 billion allocations * 40 applications * 59 thousand bindings * 0.6 ms). There is a huge potential that we could speed up the processing of the domain DSE program.

The $4.97 * 10^6$ billion seconds is too long to explore. To make the project problem addressable, we resize our design space by limiting the number of platform allocations and bindings. In our project, we discuss 600 and 6000 allocations with a maximal 100 bindings for each application.

In the project, there are three parts can be implemented into parallel processing. The first part is 40 applications analysis parallelism. The second part is the parallel of different platform allocations, which are needed to be assessed their fitness (platform performance) for the domain. The third parallel part is inside each platform assessment, which contains a large number of evaluations of 40 applications with 100 bindings. The following subsections discuss these parallel benefits, which are applied OpenMP, MPI and GPU processing.

## 4.1 OpenMP Parallelism for Domain DSE

I implement and run the OpenMP parallel domain DSE program on NEU Discovery machine. The discovery CPU model is Intel(R) Xeon(R) CPU E5-2680 at 2.8GHz, and there are a total of 20 cores (10 cores in 2 separate socketed physical CPUs).

After git pull or upload the project PDF on the Discovery, you can compile and run the program at the $PDF/$ directory using the following commands. You can modify the number of threads at line 61, 135, 142 in $PDF/ddf/eval/gpu/gpu\_cmp.cc$ for analysis, allocation and evaluation parallelism.

```
$ make ddf
$ srun ./build/release/Linux/bin/sdf3gpuCmp-ddf --appDir ./ddf/testbench/OpenVX/
    xml/ --funcDir ./ddf/testbench/OpenVX/funcDecompose/xml/ --bTable ./ddf/
    testbench/OpenVX/bindingTable/OpenVX_BindingTable.txt --aTable ./ddf/
    testbench/OpenVX/accArea/OpenVX_AccArea.txt --output ./ddf/testbench/Result/
    gpu
```

In the make ddf compilation, the OpenMP compile flag is added in the variable CXXFLAGS in file $PDF/etc/Makefile.inc$. The compilation fails sometimes on the Discovery login node because

the *xml* library is missed. To solve this problem, you can launch a compute node and compile the program, e.g. using the following commands.

```
$ srun --pty --export=ALL --partition short --tasks-per-node 1 --nodes 1 --mem
    =10Gb --time=00:30:00 /bin/bash
$ make ddf
```

Domain DSE program PDF's hierarchical parallelism is shown in the below code. The first parallelism part is 40 applications analysis. The second parallelism part is evaluation (assessment) of platform allocations, which could be parallelized at different positions, e.g. at allocations, applications, and bindings.

```
// Analysis Parallelism Position
for 40 applications {
    analysis;
}

// Evaluation Parallelism at Allocations
for 600-6000 platform allocations {
    // Evaluation Parallelism at Applications
    for 40 applications {
        for 1-100 bindings {
            evaluation;
        }
    }
}
```

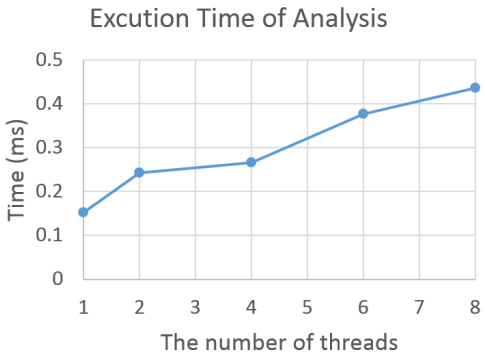
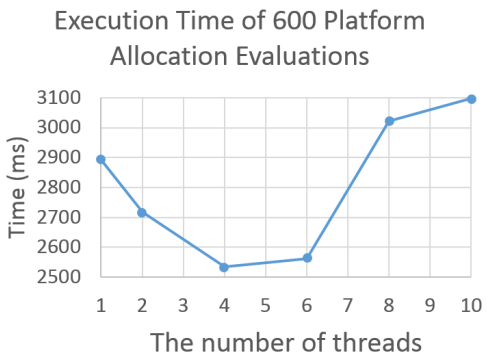



Figure 5: Analysis Execution Time

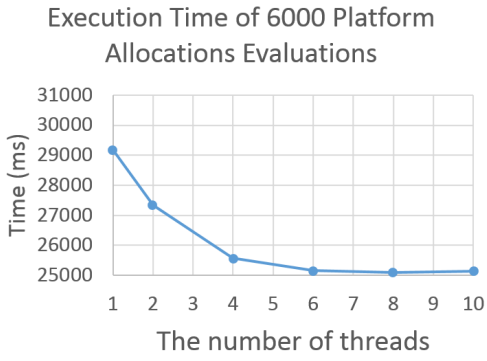
**Analysis Parallelism.** The analysis of 40 applications will obtain all OpenVX functions, and these functions will be used to generate all ACC candidates. To speed up the analysis processing, the analysis can be parallelized using OpenMP, and each thread analysis a small number of applications (40 / the number of threads).

Fig. 5 demonstrates the execution time of analysis, which is parallelized using OpenMP with a different number of threads. With the number of threads increasing, the analysis processing becomes slow instead of speedup. The reason is that the workload of application analysis is really small, and its processing execution is fast without parallelism, which is 0.152 milliseconds. The overhead of thread operations, e.g. thread creation, load, and join, is more time consuming than the analysis processing. As a result, the analysis processing doesn't need to be paralleled.

**Evaluation Parallelism at Allocations.** After the domain application analysis, we obtain different ACC candidates to accelerate the processing of functions in the domain. Due to the area and energy limitation, the platform cannot allocate all ACC candidates. Within the area budget, we need to explore a large number of platform allocations to find the optimal allocation. For each allocation, a complex evaluation is needed to assess its performance considering multiple applications and different bindings. To speed up the evaluation processing, we can parallel the evaluations using OpenMP at the loop of multiple allocations.




(a) 600 Platform Allocations




(b) 6000 Platform Allocations

Figure 6: Execution Time: Evaluation Parallelism at Allocations

In this project, 600 and 6000 allocations have been studied for evaluation parallelism. Fig. 6 demonstrates the execution time of evaluation with the increasing number of threads. With 600 allocations in Fig. 6a, 4 and 6 threads parallelism achieve the fastest processing because the workload has been distributed into multiple threads. With the number of threads increasing larger than 6, the processing time increases. The reason is that the benefit of workload distribution increases little, and the overhead of thread generation becomes dominated.

While in Fig. 6b (6000 allocations), the evaluation processing latency keeps reducing with the increasing number of threads. Each thread has more workload in 6000 allocations than 600 allocations. The benefit of workload distribution dominates the performance other than thread overhead within 10 threads. It indicates that we could parallel and speed up the evaluation using all available threads in the real domain DSE with billions of allocations.

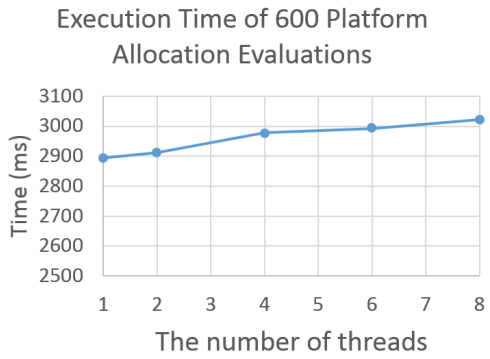



Figure 7: Execution Time: Evaluation Parallelism at Applications

**Evaluation Parallelism at Applications.** Other than parallel the evaluation at allocations, we could also parallel inside each allocation evaluation, e.g. at the for loop of multiple applications. The loop of multiple applications is lower level parallelism compared with at allocations. As Fig. 7 shown, the evaluation of 600 allocations doesn't speed up when we applied OpenMP with multiple threads. The reason is that the workload of one allocation evaluation is not large enough to gain speedup by dividing and distributing into multiple threads. According to [9], the execution time of one platform evaluation is only 0.6 milliseconds. Similar to the analysis parallelism, we conclude that the OpenMP overhead may consume the parallelism benefit of the processing with a workload of less than 1 millisecond.

Compared with parallelism at allocations in the previous, parallelism at applications has 600x more thread generations and joins. If ignoring the critical section, we should parallel the processing at the highest level to provide more workload for each thread and reduce the overhead of thread generation.

## 4.2 Multiple Processes Parallelism at Allocations

According to Section 4.1, the right position to parallel the program is at allocations in evaluation. This section discusses its parallelism using multiple processors.

In the beginning, we try to parallel the allocations using MPI. We could compile the MPI cpp program by setting the variable CXX as mpic++ at line 67 in file $PDF/etc/Makefile.inc$. However, there is a lot of information that needs to be broadcast into every processing node to do the evaluation, e.g. application structure, domain functions, ACC area, binding performance. This information is stored in complex object classes, and build MPI datatype according to the object memory layout is time-consuming.

To avoid the information transfer between processes, we ask each process to read the information and produce the output (the performance of each platform allocation) as a text file. To collect separate outputs, we could develop another program or using MPI-IO, which is in the future work. Similar to MPI coding, we separate the workload into N, and ask each process to execute 1 of N work as following.

```
indexStart = indexNode * allocations.size() / nrNode;
indexEnd = (indexNode + 1) * allocations.size() / nrNode;
for (indexAllocation = indexStart; indexAllocation < indexEnd; indexAllocation++)
{ ...
}
```

The multiple processes parallelism program can be compiled and run using the following commands on Discovery. You could modify the number of processes and the index of current process index after –nodes and –indexN.

```
$ make ddf
$ srun ./build/release/Linux/bin/sdf3gpuCmpi-ddf --appDir ./ddf/testbench/OpenVX
    /xml/ --funcDir ./ddf/testbench/OpenVX/funcDecompose/xml/ --bTable ./ddf/
    testbench/OpenVX/bindingTable/OpenVX_BindingTable.txt --aTable ./ddf/
    testbench/OpenVX/accArea/OpenVX_AccArea.txt --output ./ddf/testbench/Result/
    gpu --nodes 4 --indexN 0 &
```

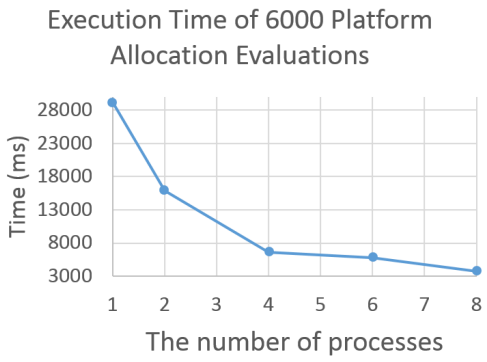



Figure 8: Evaluation Execution Time

Fig. 8 demonstrates the execution time of 6000 allocations evaluation, which is divided into N partitions and run on N processes. With the number of processes increases, each evaluation partition becomes small, and the latency of its processing reduces significantly. With 4 processes, the latency is 6646.61 milliseconds, which is 3.37x speedup compared with sequential execution using one process. While with 6 processes, there is only 4.01x speedup compared with a sequential execution.

When the number of processes smaller than 4, the latency reduction is large. While, after the number of processes larger than 4, the benefit of processing division reduces, because the workload of each process is small and there is some overhead of division. In the real domain DSE with billions of allocations, we could parallel and speed up the evaluation using many nodes.

### 4.3 GPU: Platform Evaluation Transform

The evaluation is implemented sequentially in cpp. Although the evaluation execution of each binding is fast (0.6 ms using the analytical model), it is also worth to speedup due to billions of execution, considering different allocations, applications, and bindings.

In this project, we transform the original cpp evaluation into matrix operations, which could be parallel executed multiple evaluations simultaneously using GPU. A generator is introduced to provide evaluation matrices to represent platform allocation, application binding, and function performance on different ACCs. The following commands show how to compile and run the matrix generator.

```
$ make ddf
$ srun ./build/release/Linux/bin/sdf3gpu-ddf --appDir ./ddf/testbench/OpenVX/xml
    / --funcDir ./ddf/testbench/OpenVX/funcDecompose/xml/ --bTable ./ddf/
    testbench/OpenVX/bindingTable/OpenVX_BindingTable.txt --aTable ./ddf/
    testbench/OpenVX/accArea/OpenVX_AccArea.txt --output ./ddf/testbench/Result/
    gpu
```

Then, the GPU program reads the matrices representing different allocations and application bindings, transfers these matrices into a merged data structure and copies them on GPU memory. Multiple bindings of applications on platforms are executed at the same time on GPU, and the program produces the performance (latency) of each binding. For now, our transformed GPU evaluation is a simplified version, which is only considering computation on SW and ACCs, and evaluate the processing latency. In future work, we will make our GPU evaluation more accurate considering communication, synchronization interrupt, memory size, and power consumption. You can compile and run the GPU program using the following commands.

```
$ make
$ ./eval.o
```

To compile with cpu validation of GPU output run

```
$ make debug
$ ./eval.o
```

We run the GPU program on a RTX 2080 available in the ESL lab. The program reads 1046 different bindings and runs their evaluations simultaneously. The total evaluation time is 3.6 milliseconds, which is 3.4 microseconds for each binding. Compared with the CPU evaluation time of one binding 0.6 milliseconds, GPU speeds up to around 170 times. If we increase the number of bindings simultaneously running on GPU to the maximal threads of GPU, the effective evaluation per binding will increase.

## 5 Conclusion and Future Work

In this project, we discuss the parallelism of domain DSE. After experiments, we figure out the multiple allocations in evaluation is the best position to implement parallelism using multiple threads and processes. We also transform our application evaluation into matrix operations and run them on GPU. The GPU runs multiple bindings simultaneously and achieves a significant speedup. However, due to the time limitation and coronavirus situation, there are some parts will be addressed in the future. First, combining GPU evaluation into project code. Second, providing more accurate GPU evaluation considering communication interrupts and energy consumption. Third, applying the parallelism into allocation and binding algorithms, considering domain space pruning.

# References


[1] AMD. AMD OpenVX (AMDOVX). `http://gpuopen.com/compute-product/amd-openvx/`, 2018. Accessed: 2018-02-14.

[2] Khronos Vision Working Group et al. The OpenVX Specification v1.1. *Web: https://www. khronos. org/registry/OpenVX/specs/1.1/OpenVX_Specification_1_1. pdf*, 2017.

[3] Intel. Beta for Intel Computer Vision SDK (Intel CV SDK) Support. `https://software.intel.com/en-us/computer-vision-sdk-support/code-samples`, 2017. Accessed: 2017-09-12.

[4] S. Stuijk, M. Geilen, and T. Basten. Sdf^ 3: Sdf for free. In *Application of Concurrency to System Design (ACSD)*, pages 276–278, 2006.

[5] N. Teimouri, H. Tabkhi, and G. Schirner. Alleviating scalability limitation of accelerator-based platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.

[6] J. Zhang, H. Tabkhi, and G. Schirner. DS-DSE: Domain-Specific Design Space Exploration for Streaming Applications. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 165–170, Dresden, Germany, 2018.

[7] J. Zhang, H. Tabkhi, and G. Schirner. Addressing diversity of streaming applications for allocating a unified acc-rich platform. *Northeastern University Library's Digital Repository*, 2019.

[8] J. Zhang, H. Tabkhi, and G. Schirner. Mitigating Application Diversity forAllocating a Unified ACC-Rich Platform. In *IEEE International Conference on Computer Design (ICCD)*, Abu Dhabi, UAE, 2019.

[9] J. Zhang, H. Tabkhi, and G. Schirner. Allocating one common acc-rich platform for many streaming applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.