This is the author's version of the work. The definitive work is published in *Proceedings of the 26th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Portland, OR, USA, May 17–21, 2021.*

An Efficient Approach for Image Border Handling on GPUs via Iteration Space Partitioning

Bo Qiao, Jürgen Teich, and Frank Hannig
Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
{bo.qiao, juergen.teich, frank.hannig}@fau.de

***Abstract*—Border handling is a crucial step in many image processing applications. For stencil kernels such as the Gaussian filter where a window of pixels is required to compute an output pixel, the border of the image needs to be handled differently than the body of the image. To prevent out-of-bounds accesses, conditional statements need to be inserted into the pixel address calculation. This introduces significant overhead, especially on hardware accelerators such as GPUs. Existing research efforts mostly focus on image body computations, while neglecting the importance of border handling or treating it as a corner case. In this paper, we propose an efficient border handling approach for GPUs. Our approach is based on iteration space partitioning, which is a technique similar to index-set splitting, a well-known general-purpose compiler optimization. We present a detailed systematic analysis including an analytic model that quantitatively evaluates the benefits as well as the costs of the transformation. In addition, manually implementing the border handling technique is a tedious task and not portable at all. We integrate our approach into an image processing DSL and a source-to-source compiler called Hipacc to relieve the burden and increase programmers' productivity. We evaluate over five commonly used image processing applications on two Nvidia GPUs. Results show our proposed approach achieves a geometric mean speedup of up to 87% over a naive implementation.**

I. Introduction

Image filtering is a fundamental operation in image processing, where a window of pixels is required to compute an output pixel. Filtering can also be referred to as local operators [1]. Such operators are used extensively for image smoothing, noise reduction, edge detection, etc. When computing a local operator, the image border needs to be handled differently than the body of the image. The problem arises when a part of the pixel window is accessed out of bounds. Accessing unknown memory locations may result in undefined behavior and lead to corrupted pixels. Therefore, border handling is essential to the correctness of image processing applications. Existing research efforts mostly focus on image body computations, while neglecting the importance of global border handling or treating it as a corner case. The simplest way is just to discard the corrupted image border. Nevertheless, this produces inconsistently sized images between input and output, and is unfavorable within a multi-kernel image processing pipeline. In general, border handling can be achieved either by padding additional pixels at the image border in the memory such that out of bounds accesses remain valid, or by adjusting the index address of each pixel read such that no access is out of bounds. The benefits and limitations of each approach depend on the chosen software variant as well as the underlying hardware architecture. For example, padding

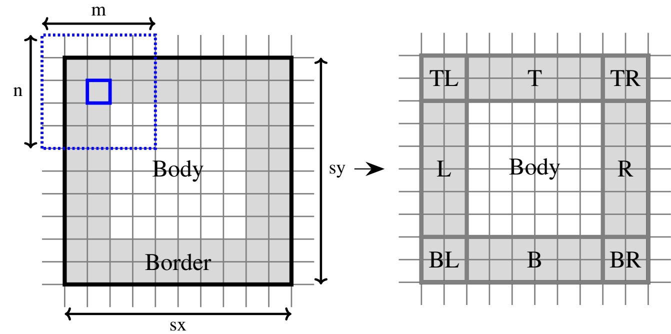
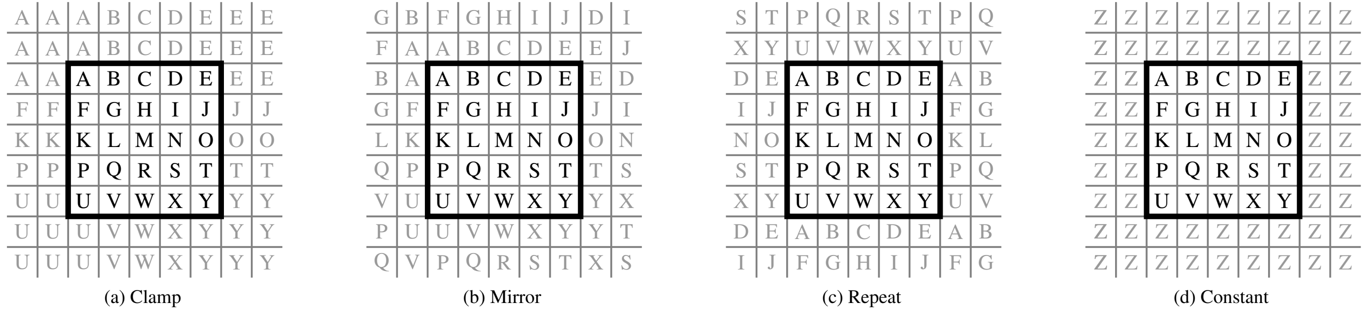


Figure 1: Border handling via iteration space partitioning.

the image border is used in most OpenCV functions [2]. One disadvantage of this approach is the required additional memory copy, which is costly, particularly for architectures such as graphics processing units (GPUs). To address this limitation, GPUs typically provide dedicated hardware support such as texture memory for global image read [3, 4]. Texture memory is cached and can be efficiently accessed at the image border. However, the access is bound to the image size and is not supported for sub-regions in the image, which makes it less flexible compared to other software-based approaches.

A naive software approach for border handling is to insert conditional statements into the address calculation of each pixel read. Although the conditional checks can guarantee no out of bounds access, they also introduce significant overhead since the statements are evaluated for every pixel read in the image. However, a closer look unveils that these checks are only needed for certain regions in the image. For example, the image body does not require any conditional statement. Therefore, we can partition the whole iteration space of the input image into multiple regions, as depicted in Figure 1. In this way, computations can be specialized for each region, depending on the relative location in the image. For example, the region on the top left (TL) only needs to be checked for the top and left border, and the region on the right (R) only needs to be checked for the right border. The main image body is the middle region (Body), where no conditional statement needs to be evaluated. We name this approach *iteration space partitioning* (ISP). The goal is to minimize the number of conditional checks required to be computed. This approach is similar to the well-known general-purpose compiler optimization technique *index-set splitting* [5].

In this paper, we present a detailed systematic analysis of ISP for image border handling on GPU architectures. We propose



(a) Clamp (b) Mirror (c) Repeat (d) Constant

Figure 2: Commonly used border handling patterns.

an analytic model to illustrate that ISP's benefits come with certain costs, and it is not always beneficial to apply ISP over a naive implementation. In addition, manually implementing ISP is a tedious task and not portable. We integrate our approach into an image processing domain-specific language (DSL) and a source-to-source compiler called Hipacc [6] to provide a complete compiler workflow that requires little implementation effort from the user. Given a desired border handling pattern as input, the Hipacc compiler is responsible for generating highly efficient GPU kernels after applying the proposed optimizations. In summary, the key contributions are:

1) A border handling approach for GPUs based on ISP.
2) An analytic model that combines domain- and architecture-specific knowledge to quantitatively evaluate the benefits as well as the costs of the ISP transformation.
3) A DSL-based automated workflow to relieve the burden of manual implementations and increase user productivity.

The remainder of this paper is organized as follows: Section II discusses the related work. Section III introduces ISP and derives the index bound used to partition the image iteration space. Section IV formulates our performance model that quantitatively evaluates the benefits and the costs of ISP. Section V presents Hipacc and the compiler workflow. The results are shown in Section VI before we conclude in Section VII.

II. Related Work

Our employed ISP approach is similar to the general-purpose compiler optimization technique called *index-set splitting* (ISS), or *loop splitting* [5, 7], where a single loop is transformed into multiple loops that iterate over disjoint portions of the original index range. This technique has inspired several existing works targeting different architectures for efficient image border management [8, 9, 10]. For field-programmable gate array (FPGA) accelerators, Bailey [8] proposed a filter design with border management to minimize hardware costs. Özkan et al. [11] proposed an analytic model that determines efficient implementations with image border handling as well as loop coarsening. Beyond image processing, border communication has been studied in scientific computing for stencil kernels, where autotuning framework by Zhang and Mueller [12] and DSLs such as StencilGen [13] and SDSL [14] provide boundary handling for the halo margins. However, these works mostly focus on reducing redundant inter-tile communications. Within the context of image processing DSLs, early efforts such as the Apply DSL [10] handles image borders by generating separate loops with different bounds-check requirements. Later, the same author proposed a declarative border handling approach with domain inference using Halide [9], a well-known image processing DSL [15]. It is highly beneficial to have an automated border handling approach within DSLs. Thus, programmers can be relieved from the burden to infer the domain size manually. Nevertheless, both mentioned works focus solely on the language constructs and do not consider GPU targets. For GPU architectures, one popular DSL that addresses border handling is Lift [16]. Like Halide, Lift is a DSL based on functional programming that offers high-order primitives such as *pad* to re-index out of bounds pixels. However, it does not further optimize the branching overhead during device execution. Another DSL that supports border handling on GPUs is Hipacc [17]. Hipacc is an open-source image processing DSL and a source-to-source compiler. It can generate CUDA and OpenCL kernels for GPU targets with efficient border handling techniques. Compared to the existing techniques in Hipacc, our approach proposes two major advancements: First, we show that it is not always beneficial to partition the iteration space during computation. The performance depends on kernel properties and inputs such as image size and block size. Based on a proposed analytic model, our approach can determine a performance-optimized implementation. Second, we illustrate that ISP can be applied at different levels of granularity. Instead of partitioning at the threadblock level, we propose to partition the iteration space at the warp level, which further improves the memory access efficiency for GPU architectures.

III. Iteration Space Partitioning

In this section, we introduce iteration space partitioning. (a) We present four commonly used patterns for image border handling. (b) We show how index-set splitting can be used in general-purpose optimizations and how it motivates our partitioning approach. (c) We derive index bounds and formulate the partitioning approach for GPU architectures.

A. Border Handling Patterns

When an image is accessed out of bounds, there are multiple patterns to compute the resulting pixels. Here, we introduce four commonly used patterns, as depicted in Figure 2: (a) *Clamp*

or *Duplicate* returns the nearest valid pixel within bounds. (b) *Mirror* returns the mirrored pixels at the border, depending on the row and column location. (c) *Repeat* or *Periodic* returns the repeated pixels at the border, which is the same as to tile the image with itself along x- and y-dimensions periodically. (d) *Constant* returns user-defined value for all out of bounds pixels. To implement these patterns, conditional checks are inserted within the address calculation of each pixel access. Assume an image of size $sx \times sy$; then, the iteration space can be defined as $x \in [0, sx), y \in [0, sy)$ and $x, y \in \mathbb{Z}$. Listing 1 depicts a C++ implementation example for these four patterns.

As can be observed in Listing 1, Clamp and Mirror implementations are similar in computational cost. Repeat uses a while loop to iteratively check the index location, which is required in the pattern when small images are computed using a large filter window. Compared to these three patterns, Constant requires a different conditional check. Unlike the other three, out of bounds pixels under the Constant pattern are independent of their original location. Hence, the pixel's value can be initialized with the user-defined constant and only be updated when pixels are within bounds.

In real-world applications, which pattern to use depends on the filtering operation as well as the expected image content [18]. For example, medical imaging applications such as multiresolution filters demand mirroring at the image border [19], while computer vision applications mostly use the clamp pattern to extend the edges in an image [20]. Nevertheless, all these four patterns require costly conditional checks before memory access, and minimizing the number of such conditional checks is the goal of iteration space partitioning. Before introducing our partitioning method, we first illustrate the concept of index-set splitting.

B. Index-Set Splitting

Index-Set Splitting (ISS) transforms a loop with conditional statements into multiple loops that iterate over disjoint portions of the original index range. An example is depicted in Listing 2. As can be seen in Listing 2, in the original loop, the conditional statements have to be executed N times for each iteration. Although the branch execution might be accurately predicted on modern architectures, the existence of such conditional statements still introduces a large overhead. Nevertheless, a closer look into the loop unveils that the first conditional statement, namely the if-clause, does not affect the computation after M iterations. Similarly, the second statement, namely the else-clause, does not affect the computation of the first M iterations. Therefore, it is possible to split the original loop into two separate loops: The first loop computes the first M iterations, and the second loop computes the remaining iterations, as can be seen in Listing 2. In this way, the previous conditional statements can be eliminated. As a result, the total number of executed instructions is reduced.

The idea can be extended to the two-dimensional iteration space in image processing: Since the conditional checks, as shown in Listing 1, do not affect the whole iteration space, we can partition the image into different regions, as depicted in Figure 1. For example, the left border check (if x < 0) only affects the left regions, namely L, TL, BL, and does not affect the other regions. Similarly, the top border check (if y < 0) only affects top regions, namely T, TL, TR. In this way, we partition the whole iteration space into multiple regions that iterate over disjoint portions of the original image. As a result, the total number of executed conditional checks can be reduced. In the following section, we derive the index bound to perform the partitioning of the iteration space.

```
T in; // current computing pixel
// (a) Clamp
if (x >= sx)  x = sx - 1;
if (y >= sy)  y = sy - 1;
if (x < 0)    x = 0;
if (y < 0)    y = 0;
in = input[x, y]; // Memory access

// (b) Mirror
if (x >= sx)  x = sx - (x + 1 - sx);
if (y >= sy)  y = sy - (y + 1 - sy);
if (x < 0)    x = 0 - x - 1;
if (y < 0)    y = 0 - y - 1;
in = input[x, y]; // Memory access

// (c) Repeat
while (x >= sx)  x = x - sx;
while (y >= sy)  y = y - sy;
while (x < 0)    x = x + sx;
while (y < 0)    y = y + sy;
in = input[x, y]; // Memory access

// (d) Constant
in = 0; // initialize with constant
if (y >= 0 && x >= 0 && y < sy && x < sx)
  in = input[x, y]; // Memory access
```

Listing 1: Example border handling implementation in C++.

```
// Original loop
for (int i = 0; i < N; i++) {
    if (i < M) { A[i] = A[i] * 2; }
    else       { A[i] = A[i] * 3; }
    B[i] = A[i] * A[i];
}

// Splitted loops, assume M < N
for (int i = 0; i < M; i++) {
    A[i] = A[i] * 2;
    B[i] = A[i] * A[i];
}
for (int i = M; i < N; i++) {
    A[i] = A[i] * 3;
    B[i] = A[i] * A[i];
}
```

Listing 2: Index-Set Splitting example in C++.

C. Partitioning on GPUs

Partitioning an image iteration space requires different strategies between CPU and GPU architectures. For CPU implementations where the iteration space is executed sequentially, it is sufficient to partition the image into body and border regions based on only the image and window size: Given an image size $sx \times sy$ and a local operator window size $m \times n$. The body region that requires no conditional checks can be defined as:

$$\{(x, y) \mid \left\lfloor \frac{m}{2} \right\rfloor \leq x < sx - \left\lfloor \frac{m}{2} \right\rfloor \wedge \left\lfloor \frac{n}{2} \right\rfloor \leq y < sy - \left\lfloor \frac{n}{2} \right\rfloor\} \quad (1)$$

The border of the image that requires conditional checks is the remaining part of the iteration space, which can be further partitioned into smaller regions. We omit the remaining derivations here since we do not consider CPUs as target architecture. After obtaining the index bound of each region, the execution is straight-forward since the computation of each region is independent of the others.

The index bound computed in Eq. (1) is not sufficient for parallel architectures such as GPUs. GPU architectures consist of an array of streaming multiprocessors (SMs) that each can execute thousands of concurrent threads. The input image is divided into threadblocks, based on a user-defined block size. Each block is dispatched by the scheduler onto the SM for execution. Therefore, the user-defined block size should be considered to partition the iteration space. Assume a user-defined block size $tx \times ty$, the index bounds can then be computed as:

$$\begin{aligned} \text{BH_L} &= \left\lceil \frac{\lfloor \frac{m}{2} \rfloor}{tx} \right\rceil, \text{BH_R} = \left\lfloor \frac{sx - \lfloor \frac{m}{2} \rfloor}{tx} \right\rfloor \\ \text{BH_T} &= \left\lceil \frac{\lfloor \frac{n}{2} \rfloor}{ty} \right\rceil, \text{BH_B} = \left\lfloor \frac{sy - \lfloor \frac{n}{2} \rfloor}{ty} \right\rfloor \end{aligned} \quad (2)$$

In Eq. (2), the four index bounds BH_L, BH_R, BH_T, BH_B collectively determine the partitioning of the iteration space. Any threadblock with an index between BH_L and BH_R in the x-dimension and between BH_T and BH_B in the y-dimension requires no border handling. The region each threadblock needs to execute can be identified by their block ID during runtime. A CUDA implementation of the routine is depicted in Listing 3.

```
if (blockIdx.x < BH_L && blockIdx.y < BH_T)
   goto TL;
if (blockIdx.x >= BH_R && blockIdx.y < BH_T)
   goto TR;
if (blockIdx.y < BH_T)
   goto T;
if (blockIdx.y >= BH_B && blockIdx.x < BH_L)
   goto BL;
if (blockIdx.y >= BH_B && blockIdx.x >= BH_R)
   goto BR;
if (blockIdx.y >= BH_B)
   goto B;
if (blockIdx.x >= BH_R)
   goto R;
if (blockIdx.x < BH_L)
   goto L;
goto Body;
```

Listing 3: Region switching for iteration space partitioning.

Instead of using the region switching statements as in Listing 3, we can also iterate over different regions in the iteration space by creating individual kernels for each region. However, this approach has two disadvantages: First is the cost of kernel launch from the host and the overhead from the additional PCIe communications. Second, programmers need to manually partition the host memory to match the original iteration space. These costs may outweigh the benefits of applying the partitioning approach. In contrast, iteration space partitioning employs one fat kernel and uses threadblock indices to control the fine-grained execution on each region during runtime. However, this approach still has two disadvantages: First, the generated fat kernel is lengthy and compromises the readability of the emitted code. Fortunately, the implementation can be largely hidden from programmers in our compiler-based approach, it should still be acceptable in most cases. Second, the additional region switching statements, as shown in Listing 3, need to be executed for all the threadblocks. This could potentially increase register usage on GPUs compared to a naive implementation. In the next section, we present an analytic model to encapsulate this trade-off.

IV. Performance Modeling

In this section, we provide a systematic analysis of the potential benefits and costs of iteration space partitioning. We use a real-world application to illustrate the benefits such as the reduction of arithmetic instructions, as well as the costs such as resource usage increase on GPUs. Finally, we propose an analytic model to encapsulate the trade-off and to suggest an optimized implementation.

A. Demystify the Benefits

By partitioning the iteration space into smaller regions, we expect the total number of conditional checks to be reduced. To understand the impact on GPU architectures, two questions need to be answered: Which instructions are reduced and in which region? And how many instructions can be reduced given an application? To answer these questions, we use a real-world application for illustration.

1) Motivation Example: Bilateral Filter: The bilateral filter is a widely used noise-removing filter that preserves edges in image processing [21]. Each output pixel depends on the neighborhood pixels of the same location. It basically performs two convolutions together, one for computing the spatial closeness component and the other one for the intensity similarity component. In general, the computation of such local operators consists of two steps: Address calculation and kernel computation. Address calculation needs to be performed for each accessed pixel in the filter window, and it is generally computed in integers. The conditional checks required for border handling are part of the address calculation, which is also where the iteration space partitioning approach intends to optimize.

We performed two CUDA implementations for the bilateral filter: A naive version without ISP, namely the conditional checks are performed for the whole iteration space, and an optimized version with border handling using ISP. The compilation was executed on a GTX680 using CUDA 10, under Ubuntu 18.04 LTS. The window size used in the filter is 13×13. The border handling pattern is Clamp. We compiled the CUDA kernel to PTX, and inventoried all executed instructions in the PTX code. We choose to measure at PTX level to obtain a more accurate estimation than at CUDA source code. Further, it guarantees portability since PTX is a machine-independent ISA across different architectures. Finally, we manually disassembled the PTX code of the optimized version to obtain the statistics

ISA	Border Handling with ISP (partioned into 9 regions)									Naive
	TL	TR	T	BL	BR	B	R	L	Body	
add	703	704	366	704	705	367	704	703	377	705
sub	169	169	169	169	169	169	169	169	169	169
mul	1027	1027	1014	1027	1027	1014	1027	1027	1026	1027
div	2	2	2	2	2	2	2	2	2	2
max	24	12	12	12	0	0	0	12	0	26
min	0	1	0	1	2	1	1	0	0	2
fma	507	507	507	507	507	507	507	507	507	507
mad	4	4	17	4	4	17	4	4	4	4
ex2	338	338	338	338	338	338	338	338	338	338
cvt	512	512	356	512	512	356	512	512	354	512
cvta	2	2	2	2	2	2	2	2	2	2
ld	170	170	170	170	170	170	170	170	169	170
st	1	1	1	1	1	1	1	1	1	1
setp	342	355	342	355	367	355	357	345	344	363
selp	340	351	339	351	362	350	350	339	338	362
mov	10	10	10	10	9	9	9	10	9	10
neg	169	169	169	169	169	169	169	169	169	169
and	1	2	2	3	4	4	4	4	4	0
Total	4321	4336	3816	4337	4350	3831	4326	4314	3813	4369

TABLE I: Bilateral filter PTX instruction comparison.

for each region. The instructions have been categorized based on keywords for simplicity purposes. For example, add.s32 and add.f32 are both counted as an add instruction. Note that the counted number of instructions includes both the kernel execution in the region as well as the additional switching statement to go to a region as shown in Listing 3. The results are depicted in Table I.

We can make two important observations from Table I: First, compared to the naive implementation, not all the regions have a noticeable reduction in the total number of executed instructions. It seems counter-intuitive that only three (T, B, Body) out of nine regions have a clear benefit over the naive approach. This can also be observed for other applications such as a Gaussian filter, which we will not show here for simplicity. One reason is that the naive version may have many conditional statements in the source code, but many of them share common sub-expressions that can be optimized by the NVCC compiler. In addition, the region switching statements also account for extra instructions for the border regions. Second, compared to the regions with clear benefits (T, B, Body), the naive implementation is more costly only for certain arithmetic instructions such as max, add, and cvt. Therefore, the benefit of iteration space partitioning mostly reduces arithmetic pipeline utilization on the GPU. Next, we construct an analytic model to capture the number of reductions.

2) Benefit Modeling: We assume the number of instructions[1] to check one border (e.g., the left border) is n_{check}, and the number of instructions to execute the actual kernel is n_{kernel}. Then, the total number of instructions to execute a naive implementation N_{naive} can be estimated as:

$$N_{\text{naive}} = (n_{\text{check}} \cdot 4 + n_{\text{kernel}}) \cdot m \cdot n \cdot sx \cdot sy \tag{3}$$

Eq. (3) is straightforward to understand: The naive implementation requires four border checks for each accessed pixel, as shown in Listing 1, within each window size $m \cdot n$, and the window slides along the whole iteration space of size $sx \cdot sy$.

[1]The type of instructions is not required in our model, since both the ISP and the Naive approach use the same checking statements.

For the ISP implementation, the total number of instructions N_{ISP} is the sum of all the regions, namely

$$N_{\text{ISP}} = \sum\nolimits_{p \in \{\text{TL, TR, T, BL, BR, B, R, L, Body}\}} n_{\text{inst}}(p) \tag{4}$$

In Eq. (4), $n_{\text{inst}}(p)$ is the number of instructions executed in region p. For each region p, $n_{\text{inst}}(p)$ can be further estimated as:

$$n_{\text{inst}}(p) = (n_{\text{switch}}(p) + n_{\text{region}}(p)) \cdot m \cdot n \cdot n_{\text{block}}(p) \cdot tx \cdot ty \tag{5}$$

Here, $n_{\text{switch}}(p)$ denotes the number of instructions needed to switch to region p, and $n_{\text{region}}(p)$ denotes the number of instructions needed to execute this region, including address calculation and kernel execution. Additionally, $n_{\text{block}}(p)$ denotes the number of threadblocks executing this region. n_{switch} can be estimated based on Listing 3. n_{region} can be estimated as follows:

$$n_{\text{region}}(p) = \begin{cases} n_{\text{check}} \cdot 2 + n_{\text{kernel}} & \text{if } p \in \{\text{TL, TR, BL, BR}\} \\ n_{\text{check}} + n_{\text{kernel}} & \text{if } p \in \{\text{T, R, L, B}\} \\ n_{\text{kernel}} & \text{if } p = \text{Body} \end{cases} \tag{6}$$

Obviously, each region only needs to execute part of the conditional check depending on its location. For example, region TL requires two border checks, while region L requires only one border check. The Body region requires no border check.

Finally, we need to determine $n_{\text{block}}(p)$ in Eq. (5), namely the number of blocks executing each region. First, we derive the total number of blocks in the x-dimension, denoted as N_{blockx}, the total number of blocks in the y-dimension, denoted as N_{blocky}, and the total number of blocks in both dimensions, denoted as N_{block}.

$$N_{\text{blockx}} = \left\lceil \frac{sx}{tx} \right\rceil, N_{\text{blocky}} = \left\lceil \frac{sy}{ty} \right\rceil, N_{\text{block}} = N_{\text{blockx}} \cdot N_{\text{blocky}} \tag{7}$$

Then, based on the index bounds derived earlier in Eq. (2), the number of blocks executing each region can be computed as:

$$n_{\text{block}}(p) = \begin{cases} (\text{BH_R} - \text{BH_L}) \cdot \text{BH_T} & \text{if } p = \text{T} \\ (\text{BH_R} - \text{BH_L}) \cdot (N_{\text{blocky}} - \text{BH_B}) & \text{if } p = \text{B} \\ \text{BH_T} \cdot \text{BH_L} & \text{if } p = \text{TL} \\ \text{BH_T} \cdot (N_{\text{blockx}} - \text{BH_R}) & \text{if } p = \text{TR} \\ (N_{\text{blocky}} - \text{BH_B}) \cdot \text{BH_L} & \text{if } p = \text{BL} \\ (N_{\text{blocky}} - \text{BH_B}) \cdot (N_{\text{blockx}} - \text{BH_R}) & \text{if } p = \text{BR} \\ \text{BH_L} \cdot (\text{BH_B} - \text{BH_T}) & \text{if } p = \text{L} \\ (N_{\text{blockx}} - \text{BH_R}) \cdot (\text{BH_B} - \text{BH_T}) & \text{if } p = \text{R} \end{cases} \tag{8a}$$

$$n_{\text{block}}(\text{Body}) = N_{\text{block}} - \sum\nolimits_{p \in \{\text{TL, TR, T, BL, BR, B, R, L}\}} n_{\text{block}}(p) \tag{8b}$$

As can be seen in Eq. (8a), the number of blocks executing each region is determined by the window size, block size, and image size. The partitioning is performed to minimize the number of blocks that execute border regions, hence to maximize the number of blocks that execute the body region as shown in Eq. (8b). Finally, we can provide the estimated number of instructions for both implementations and the ratio of the two as:

$$R_{\text{reduced}} = \frac{N_{\text{naive}}}{N_{\text{ISP}}} \tag{9}$$

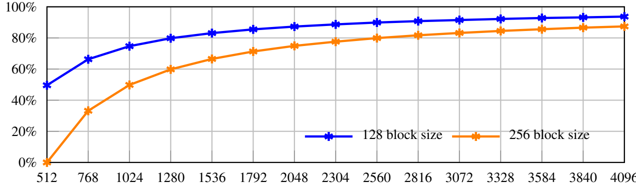


Figure 3: Percentage of blocks executed in region Body.

3) Observations from the Model: Two observations can be made from our derived analytical model: First, if the kernel computation n_{kernel} is relatively small compared to the address calculation n_{check}, the iteration space partitioning is likely to contribute a larger reduction of instructions. On the other hand, expensive kernels are likely to benefit less since the address calculation is insignificant. Second, as suggested by the model, the amount of instruction reduction depends on the image size as well as the user-defined block size, which in turn determines how many blocks execute which region. Based on Eq. (8a) and Eq. (8b), large images are likely to benefit more from ISP due to the high percentage of blocks executed in the body region. To illustrate this relation, we computed the percentage of blocks that execute the body region for a 5×5 local operator with two different block size configurations, as depicted in Figure 3. The x-axis denotes the image size on one dimension, and all images have the same size in both x- and y-dimension. As can be seen, given a block size, smaller images have a lower percentage of blocks executing the body region. When small images are computed using a large block size, there are not many blocks left to execute the body region. In this case, the overall performance of ISP might be worse than a naive implementation. Next, we incorporate a cost parameter into the model.

B. Cost Model

Although iteration space partitioning can reduce the number of executed instructions, it comes with a certain cost such as resource usage increase. The region switching statements before partitioning may introduce additional register usage. To illustrate the effect of resource increase, we benchmarked the Bilateral filter using all four border handling patterns under the same environment as the previous PTX code generation. Then, we computed the speedups of the ISP implementation over the naive implementation. The results are depicted in Figure 4. As can be seen in the figure, ISP is not always beneficial. For small images, e.g., size of 512×512, speedups of Mirror, Clamp, and Constant implementations are less than 1.0, which implies

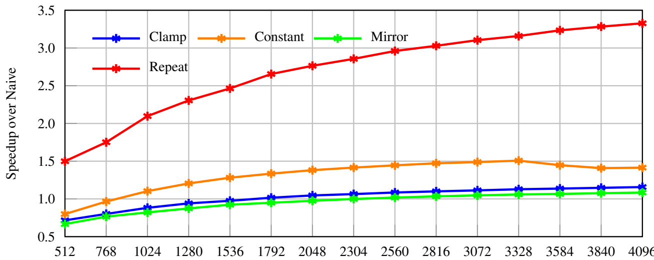


Figure 4: Bilateral filter speedups over naive implementation.

	Implem.	Clamp	Mirror	Repeat	Constant
register usage	naive ISP	32 40	32 40	18 32	32 63
occupancy (theor.)	naive ISP	100% 75%	100% 75%	100% 100%	100% 50%

TABLE II: Bilateral filter register usage and occupancy.

that the naive approach with all conditional checks is the better implementation. This is due to increase of register usage that further reduces the achieved occupancy.

1) Reduced Occupancy: We logged the register usage for each implementation and computed the theoretical occupancy for the underlying architecture. The results are summarized in Table II. As can be seen, compared to the naive implementation, ISP increases the register usage under all four border handling patterns. Moreover, three of the four patterns eventually result in a decrease of the computed theoretical occupancy. For GPU architectures, there exists a maximum number of threadblocks or warps that can be concurrently active on an SM [22]. In addition to this hardware limit, the number of threadblocks that can be executed concurrently also depends on the hardware resource usage, namely user-defined block size, register usage, etc. When a certain resource is heavily used in the kernel, fewer blocks will be executed concurrently. Since the total number of blocks is fixed for the given image and block size, the device will need more rounds/iterations to process all the blocks. This increase in the number of rounds/iterations can be modeled by an increase in the total number of instructions that need to be executed. In other words, a reduction in occupancy on the device can be modeled by an increase in the total number of instructions being executed. Next, we put all these together to formulate a prediction model.

2) Prediction Model: Assume the obtained theoretical occupancy is O_{naive} and O_{ISP} for both implementations. If the amount of occupancy decreases from O_{naive} to O_{ISP}, the number of rounds to execute all the blocks is increased by $O_{\text{naive}}/O_{\text{ISP}}$. Combined with the previous derived amount of instruction reduction in Eq. (9), we can derive a prediction model as:

$$G = \frac{N_{\text{naive}}}{N_{\text{ISP}} \cdot O_{\text{naive}}/O_{\text{ISP}}} = R_{\text{reduced}} \cdot \frac{O_{\text{ISP}}}{O_{\text{naive}}} \tag{10}$$

	Clamp		Constant		Mirror		Repeat	
Img Sz	meas.	pred.	meas.	pred.	meas.	pred.	meas.	pred.
512	Naive	Naive	Naive	Naive	Naive	Naive	ISP	ISP
768	Naive	Naive	Naive	Naive	Naive	Naive	ISP	ISP
1024	Naive	Naive	ISP	Naive	Naive	Naive	ISP	ISP
1280	Naive	Naive	ISP	Naive	Naive	Naive	ISP	ISP
1536	Naive	ISP	ISP	Naive	Naive	ISP	ISP	ISP
1792	ISP	ISP	ISP	ISP	Naive	ISP	ISP	ISP
2048	ISP	ISP	ISP	ISP	Naive	ISP	ISP	ISP
2304	ISP	ISP	ISP	ISP	Naive	ISP	ISP	ISP
2560	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
2816	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
3072	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
3328	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
3584	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
3840	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
4096	ISP	ISP	ISP	ISP	ISP	ISP	ISP	ISP
Pears.	0.9791		0.9792		0.9798		0.9817	

TABLE III: Measurement vs. model prediction.

Eq. (10) predicts the potential gains of a ISP implementation over the naive implementation. It captures the trade-off between instruction reduction and occupancy reduction. If G is larger than 1, it predicts the ISP implementation to be faster; otherwise, the naive implementation should be used. Table III summarizes the measured best implementations as well as our model-predicted results for the Bilateral application. Green colored items indicate a good prediction by our model since it is the same as the measured ground truth. Red colored items indicate a misprediction since it deviates from the measurement. In addition, we computed the Pearson correlation coefficient for each border handling pattern to indicate the amount of correlation between the output value from the model and the measurement. As can be seen in Table III and Figure 4, the model has only a few mispredictions around the switching point between the two implementations, where the performance difference is insignificant. For small and large images where the speedup or slowdown is significantly higher, our model can deliver an accurate prediction.

C. Intermediate Summary

We have presented a systematic analysis of the iteration space partitioning approach. This optimization can reduce the number of executed instructions in the address calculation, but at the increasing cost of register usage. We constructed an analytic model to encapsulate this trade-off with the help of domain knowledge such as mask size and architecture knowledge such as occupancy. In the end, the model delivers a good suggestion of which implementation is the best option. However, manual implementations of the ISP transformation are tedious, error-prone, and not portable. Therefore, we present an integration of the proposed approach into an image processing DSL to enable automatic code generation.

V. DSL-Based Implementation

The tooling we employed in this work is Hipacc, an open-source image processing DSL embedded into C++, and a source-to-source compiler based on Clang/LLVM to generate CUDA and OpenCL code for GPUs. In contrast to DSLs based on functional programming such as Halide, Hipacc offers powerful domain-specific primitive operators to the user. Such operators provide abstractions for common compute patterns in image processing. Stencil kernels are called local operators in Hipacc. Such operators can be specified by implementing a user-defined class. The class is derived from the *Kernel* class in the Hipacc DSL, which provides a virtual *kernel()* function to be implemented in the user class. A kernel implementation of the Bilateral filter and the host code to launch the kernel in Hipacc are depicted in Listing 4. In the kernel implementation, *dom* indicates the sliding window of the filter, and *iterate* applies the filter computation. To launch the kernel from the host code, first, a mask is defined with pre-computed coefficients. The domain can be automatically inferred from the mask. Then, the input image can be read and assigned to the DSL *image* object. After that, a border handling pattern can be specified using the *BoundaryCondition* object. Afterward, the output image and its iteration space can be defined. Finally, the kernel can be instantiated and executed.

```
// kernel implementation
class BilateralFilter : public Kernel<T> {
  // ...
  void kernel() {
    float d = 0, p = 0;
    iterate(dom, [&] () -> void {
      float diff = in(dom) - in();
      float c = mask(dom);
      float s = expf(-c_r * diff*diff);
      d += c*s;
      p += c*s * in(dom);
    });
    output() = p/d;
  }
};

// host kernel launch
// mask and domain with pre-computed coefficients
Mask<T> mask(coef);
Domain dom(mask);

// input image
T *image = read_image(...);
Image<T> in(width, height, image);
// clamping as boundary condition
BoundaryCondition<T> bound(in,dom,Boundary::CLAMP);
Accessor<T> acc(bound);

// output image
Image<T> out(width, height);
IterationSpace<T> iter(out);

// instantiate and launch the Bilateral filter
BilateralFilter filter(iter,acc,mask,dom,sigma_r);
filter.execute();
```

Listing 4: Bilateral filter in Hipacc.

As can be seen in Listing 4, programmers can focus on the filter computation and do not have to worry about any index calculations. The source-to-source compiler is responsible for the lowering and optimizations, including iteration space partitioning. In this section, we first briefly describe the compiler workflow in Hipacc. Then, we explain how to further optimize the ISP by implementing it using *warp-grained partitioning*.

A. Hipacc Workflow

Figure 5 depicts an overview of the Hipacc tool flow that translates the simple user description as depicted in Listing 4 to a CUDA implementation. Using the Hipacc DSL, programmers specify their applications in C++, which can be parsed by Clang to generate the Clang abstract syntax tree (AST). Then, the Hipacc compiler traverses the AST and performs the source-to-source transformation using two internal libraries: *Analyze* gathers information for analysis and optimizations, e.g., domain

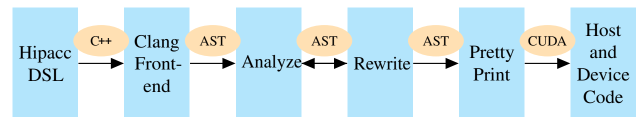


Figure 5: Hipacc workflow.

knowledge such as the compute pattern, operator window size, kernel register usage, kernel data dependence, or device information such as the compute capability and number of available registers. Based on the obtained knowledge, optimizations such as kernel fusion or concurrent kernel execution are exploited [23, 24, 25, 26]. After the analysis, the transformations and code generations are performed by the other library *Rewrite*. It uses the runtime API to launch the kernels. In the end, the optimized implementation is pretty-printed to host and device code.

B. Warp-Grained Partitioning

On GPU architectures, a threadblock is the basic unit for kernel execution from a programmer's perspective. Nevertheless, after each block is dispatched onto the SM, it is further divided into warps. A warp is a set of 32 threads that performs single instruction multiple thread (SIMT) execution in lock steps. Consider a block size of 128, and it consists of 4 warps, and a local operator with a small window size such as 3×3. Then, in the x-dimension, only the leftmost warps require the left border check. Similarly, only the rightmost warps require the right border check. The analysis here is analogous to the block level described in previous sections. Therefore, it is possible to further improve the partitioning granularity by adding the warp location into our region switching statement. Listing 5 depicts a refined region switching method with the help of the warp location. As can be seen, similar to the previously computed block index bound BH_L and BH_R, the warp index bound are denoted by W_L and W_R in the x-dimension. We only consider the warp location in the x-dimension because the block layout in GPU applications is mostly wide in x-dimension, which uses memory more efficiently. Recall our analysis based on Table I, we would like more blocks to execute the T, B, and Body region due to their significant reduction gains. With the help of the warp index, we can switch the executions of some expensive regions to cheaper ones.

```
if (blockIdx.x < BH_L && blockIdx.y < BH_T)
  if (warpID.x > W_L) goto T;
  goto TL;
if (blockIdx.x >= BH_R && blockIdx.y < BH_T)
  if (warpID.x < W_R) goto T;
  goto TR;
if (blockIdx.y < BH_T)
  goto T;
if (blockIdx.y >= BH_B && blockIdx.x < BH_L)
  if (warpID.x > W_L) goto B;
  goto BL;
if (blockIdx.y >= BH_B && blockIdx.x >= BH_R)
  if (warpID.x < W_R) goto B;
  goto BR;
if (blockIdx.y >= BH_B)
  goto B;
if (blockIdx.x >= BH_R)
  if (warpID.x < W_R) goto Body;
  goto R;
if (blockIdx.x < BH_L)
  if (warpID.x > W_L) goto Body;
  goto L;
goto Body;
```

Listing 5: Region switching with warp index.

VI. Evaluation and Results

In this section, we present the results of benchmarking five commonly used filters in image processing: Gaussian, Laplace, Bilateral, Sobel, and Night filter [27]. Gaussian, Laplace, and Bilateral are single kernel implementations with a window size of 3×3, 5×5, and 13×13, respectively. The Sobel filter consists of 3 kernels to compute x-, y-derivatives, and the magnitude, among which the first two are local operators. The night filter consists of five kernels that first iteratively applying the Atrous (with holes) algorithm [28] with different sizes (3×3, 5×5, 9×9, 17×17), before performing the actual tone mapping. For each application, we benchmarked all four border handling patterns, namely Clamp, Mirror, Repeat, and Constant, using four image sizes 512×512, 1024×1024, 2048×2048, and 4096×4096. We chose these implementation variants to cover a mixture of different window sizes, number of kernels, and the expensiveness of the filter kernel computation. The employed GPUs are Nvidia GTX680 and RTX2080, with Kepler and Turing architecture, respectively. The host OS is Ubuntu 18.04 LTS with CUDA version 11.0. The execution time is obtained from the output of NVProf. We computed the speedups of the *isp* implementation (always apply ISP) over the naive implementation, as well as the *isp+m* implementation (apply ISP based on model prediction) over the naive implementation. The results are gathered in Figure 6.

A. Discussions

1) Always Apply ISP over Naive Implementation: As can be observed in Figure 6, the isp approach contributes a speedup over the naive implementation in most cases. Especially for large image sizes such as 2048×2048 and 4096×4096. In general, the amount of speedup also increases with the computed image size, which has been analyzed by our model, as discussed in Figure 3. Furthermore, the Repeat border handling pattern benefits more from the ISP approach than the other three patterns, due to the more costly address calculation. For example, the Gaussian kernel achieves a speedup of up to 2.9 for the Repeat pattern and up to 1.3 for the others, as can be seen in subplots ① to ⑧.

2) Apply ISP based on Model Prediction: As analyzed in our prediction model, it is not always beneficial to apply the ISP optimization for certain applications such as the Bilateral filter. This can be seen in subplot ⑰ (Bilateral-Clamp-GTX680) and ⑱ (Bilateral-Mirror-GTX680), where the 512×512 image has a speedup of less than 1.0 for the ISP implementation. In these scenarios, our prediction model suggests falling back to the naive implementation. In these cases, applying ISP would slowdown the execution by about 40%.

In only a few scenarios, our model did not predict the implementation performance accurately. For example, on the RTX2080 when executing the bilateral filter with small images, our model always suggested the ISP implementation, as can be seen in subplot ㉑ and ㉒. The model did not see a reduction in occupancy in these scenarios due to the increased number of available registers on the Turing architecture. As a result, the implementation is up to 30% slower than the naive version. Nevertheless, by combining ISP with the prediction model, we

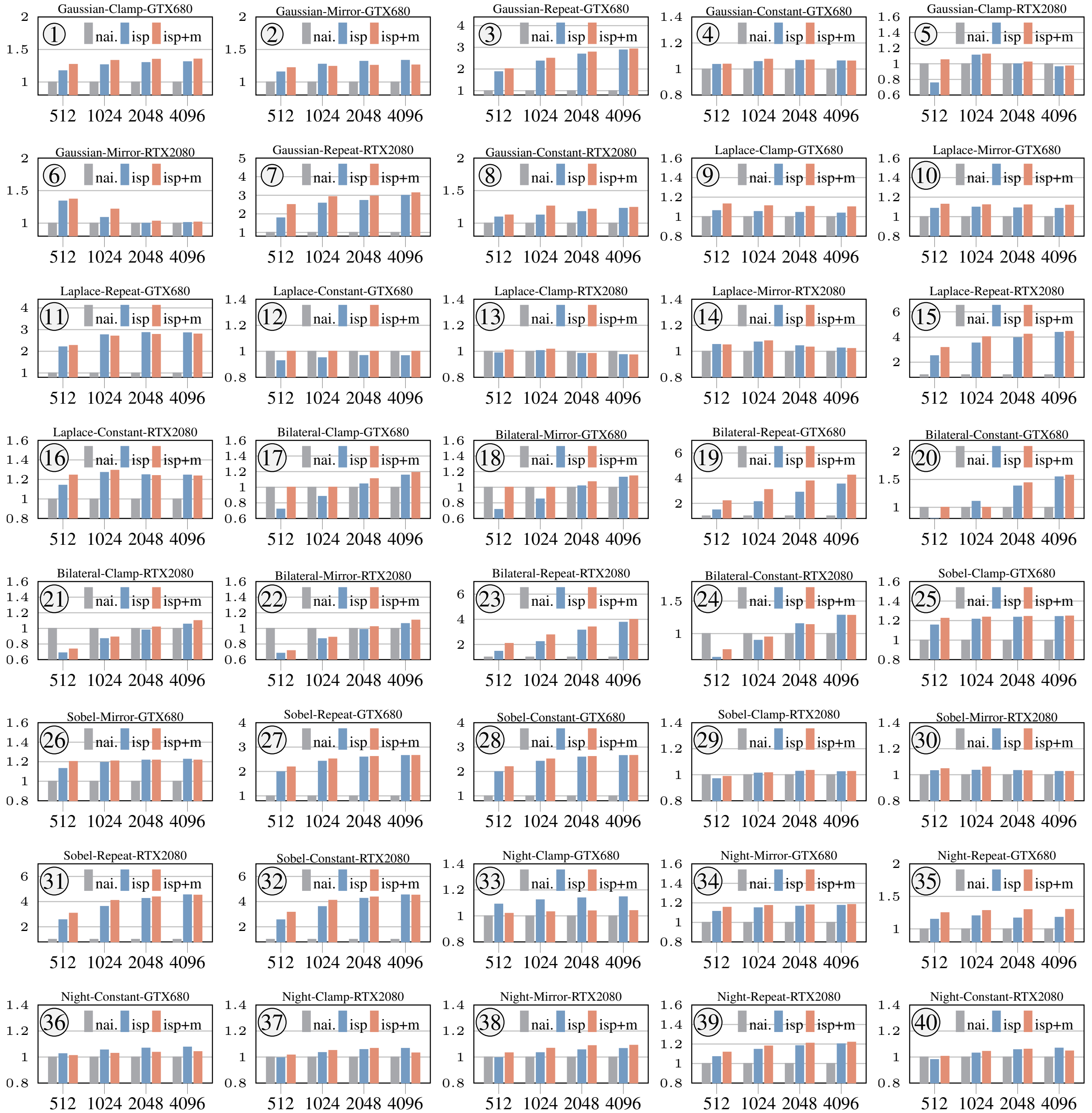


Figure 6: Normalized speedups: naive (nai.), always apply ISP (isp), apply ISP based on model prediction (isp+m).

can achieve a performance-optimized implementation in almost all scenarios.

3) Geometric Mean Speedups: For each application, we computed the geometric mean of the speedups of the isp+m implementation over the naive implementation across all benchmarks on both GPUs. The result is given in Table IV. Our final implementation can achieve a speedup range from 10% to 87%. In general, we can observe that the less expensive the kernel computation is, the more speedup our approach contributes. As can be seen in the table, Gaussian and Laplace have a higher geometric speedup (1.438, 1.422) than Bilateral and Night filter (1.355, 1.102). Our approach benefits most for applications such as the Sobel filter that consists of multiple less expensive kernels, where a speedup of more than 4.0 can be achieved on the RTX2080.

over Naive	Gaussian	Laplace	Bilateral	Sobel	Night
Geo. Mean	1.438	1.422	1.355	1.877	1.102

TABLE IV: Geometric mean across all implementations.

VII. Conclusion

We proposed an efficient approach for image border handling on GPUs. Conditional checks that are needed for address calculations in stencil kernels introduce huge overhead, which can be optimized by partitioning the iteration space into disjoint regions. Although partitioning can reduce the number of executed conditional statements, it also comes with additional cost in register increase. We showed that it is possible to encapsulate this trade-off using an analytic model and make accurate predictions based on the combination of domain- and architecture-specific knowledge. We also integrated the proposed approach into a source-to-source compiler called Hipacc, which enables a complete workflow to generate efficient kernels for GPUs automatically. The results showed our approach could achieve a geometric mean speedup of 87% over a naive implementation without efficient border handling. As a next step, we plan to explore the ISP optimization on irregular stencil kernels beyond image processing, such as using a sparse stencil mask that is only applied to a few neighbors.

Acknowledgments: This work has been partially supported by Siemens Healthineers AG, Erlangen, Germany.

References

[1] I. N. Bankman. *Handbook of Medical Image Processing and Analysis*. Vol. 2. Academic Press, Dec. 2008. ISBN: 978-0-123-73904-9.

[2] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. 2nd. O'Reilly Media, Inc., 2013. ISBN: 1449314651.

[3] Nvidia. *CUDA C++ Programming Guide*. 2020. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-fetching.

[4] Advanced Micro Devices. *OpenCL Programming Guide*. 2020. URL: https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html.

[5] M. J. Wolfe, C. Shanklin, and L. Ortega. *High Performance Compilers for Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[6] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. "HIPAcc: A Domain-Specific Language and Compiler for Image Processing". In: *IEEE Transactions on Parallel and Distributed Systems* (2016). DOI: 10.1109/TPDS.2015.2394802.

[7] R. Sakellariou. *On the Quest for Perfect Load Balance in Loop-based Parallel Computations*. Tech. rep. UMCS-TR–98-2-1. University of Manchester. Department of Computer Science and University of Manchester. Faculty of Education, 1997.

[8] D. G. Bailey. "Image Border Management for FPGA Based Filters". In: *Proceedings of the Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)*. 2011. DOI: 10.1109/DELTA.2011.34.

[9] L. G. C. Hamey. "A Functional Approach to Border Handling in Image Processing". In: *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*. 2015.

[10] L. G. C. Hamey. "Efficient Image Processing with the Apply Language". In: *9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA)*. 2007, pp. 533–540.

[11] M. Özkan, O. Reiche, F. Hannig, and J. Teich. "Hardware Design and Analysis of Efficient Loop Coarsening and Border Handling for Image Processing". In: *Proceedings of the 28th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. DOI: 10.1109/ASAP.2017.7995273.

[12] Y. Zhang and F. Mueller. "Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters". In: *IEEE Transactions on Parallel and Distributed Systems* 24.3 (2013), pp. 417–427. DOI: 10.1109/TPDS.2012.160.

[13] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan. "Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1902–1920. DOI: 10.1109/JPROC.2018.2862896.

[14] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. "SDSLc: A Multi-Target Domain-Specific Compiler for Stencil Computations". In: *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. WOLFHPC '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340168. DOI: 10.1145/2830018.2830025.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Seattle, WA, USA). 2013. DOI: 10.1145/2491956.2462176.

[16] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. "High Performance Stencil Code Generation with Lift". In: *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. Vienna, Austria: ACM, 2018, pp. 100–112. DOI: 10.1145/3168824.

[17] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. "Generating Device-specific GPU Code for Local Operators in Medical Imaging". In: *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Shanghai, China). May 21–25, 2012. DOI: 10.1109/IPDPS.2012.59.

[18] D. G. Bailey and A. S. Ambikumar. "Border Handling for 2D Transpose Filter Structures on an FPGA". In: *J. Imaging* 4 (2018), p. 138. DOI: 10.3390/jimaging4120138.

[19] D. Kunz, K. Eck, H. Fillbrandt, and T. Aach. "Nonlinear Multiresolution Gradient Adaptive Filter for Medical Images". In: *Proc. SPIE* 5032 (Feb. 2003). DOI: 10.1117/12.481323.

[20] X. Tan and B. Triggs. "Enhanced Local Texture Feature Sets for Face Recognition Under Difficult Lighting Conditions". In: *IEEE Transactions on Image Processing* 19.6 (2010), pp. 1635–1650.

[21] C. Tomasi and R. Manduchi. "Bilateral Filtering for Gray and Color Images". In: *Proc. of the 6th Int'l Conference on Computer Vision*. Jan. 1998, pp. 839–846. DOI: 10.1109/ICCV.1998.710815.

[22] Nvidia. *Achieved Occupancy*. 2020. URL: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[23] B. Qiao, O. Reiche, F. Hannig, and J. Teich. "From Loop Fusion to Kernel Fusion: A Domain-Specific Approach to Locality Optimization". In: *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA). 2019. DOI: 10.1109/CGO.2019.8661176.

[24] B. Qiao, O. Reiche, J. Teich, and F. Hannig. "Unveiling Kernel Concurrency in Multiresolution Filters on GPUs with an Image Processing DSL". In: *Proceedings of the 13th Annual Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU)* (San Diego, CA, USA). 2020. DOI: 10.1145/3366428.3380773.

[25] B. Qiao, M. A. Özkan, J. Teich, and F. Hannig. "The Best of Both Worlds: Combining CUDA Graph with an Image Processing DSL". In: *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC)* (San Francisco, USA). IEEE, 2020. ISBN: 9781450367257. DOI: 10.1109/DAC18072.2020.9218531.

[26] B. Qiao, O. Reiche, M. A. Özkan, J. Teich, and F. Hannig. "Efficient Parallel Reduction on GPUs with Hipacc". In: *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. SCOPES '20. St. Goar, Germany, 2020, pp. 58–61. DOI: 10.1145/3378678.3391885.

[27] H. W. Jensen, S. Premoze, P. Shirley, W. B. Thompson, J. A. Ferwerda, and M. M. Stark. *Night Rendering*. Tech. rep. UUCS-00-016. Computer Science Department, University of Utah, Aug. 2000.

[28] M. J. Shensa. "The Discrete Wavelet Transform: Wedding the À trous and Mallat algorithms". In: *IEEE Transactions on Signal Processing* 40.10 (Oct. 1992), pp. 2464–2482.