From Loop Fusion to Kernel Fusion: A Domain-Specific Approach to Locality Optimization

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

***Abstract*—Optimizing data-intensive applications such as image processing for GPU targets with complex memory hierarchies requires to explore the tradeoffs among locality, parallelism, and computation. Loop fusion as one of the classical optimization techniques has been proven effective to improve locality at the function level. Algorithms in image processing are increasing their complexities and generally consist of many kernels in a pipeline. The inter-kernel communications are intensive and exhibit another opportunity for locality improvement at the system level. The scope of this paper is an optimization technique called kernel fusion for data locality improvement. We present a formal description of the problem by defining an objective function for locality optimization. By transforming the fusion problem to a graph partitioning problem, we propose a solution based on the minimum cut technique to search fusible kernels recursively. In addition, we develop an analytic model to quantitatively estimate potential locality improvement by incorporating domain-specific knowledge and architecture details. The proposed technique is implemented in an image processing DSL and source-to-source compiler called Hipacc, and evaluated over six image processing applications on three Nvidia GPUs. A geometric mean speedup of up to 2.52 can be observed in our experiments[1].**

I. Introduction

Data-intensive applications such as image processing gain more and more importance across multiple domains including computer vision, medical imaging, machine learning, and autonomous driving. The intrinsic data parallelism of such applications demands for heterogeneous architectures with accelerators such as graphics processing units (GPUs) for execution. Nevertheless, modern parallel computing architectures exhibit complex memory hierarchies in a variety of ways. Close-to-compute memory is generally fast but scarce, and might not be shared globally such as registers on GPUs. Whereas large, global memory is often costly to access. Achieving good performance requires efficient memory management, and incurs the tradeoffs among locality, parallelism, and the number of redundant computations, as depicted in Figure 1.

Loop transformations have been studied extensively to best exploit the tradeoffs. Example techniques are loop fusion and loop tiling [1] [2], which target at locality optimization and have been proven effective in multiple domains such as linear algebra, machine learning, and image processing [3] [4]. The basic idea is to aggregate multiple loops into one, reduce the reuse distance, and increase the opportunity for small intermediate data to reside in cache. Loop fusion is typically one of the first steps among the loop optimizations since it leverages the optimization scope and increases the opportunity for parallelism and locality exploration [5]. Nevertheless, fusion remains a challenging problem constrained from data dependence and resources [6]. One method to search fusible candidates is by greedy fusion, namely fusing along the heaviest edge. Mullapudi et al. [7] presented PolyMage, an image processing domain-specific language (DSL) and compiler that performs automatic fusion and tiling of image processing pipelines. The grouping step in their algorithm is essentially a pair-wise greedy fusion, expanding the fusion scope while accounting for the fusion profitability. Halide's auto-scheduler employs a similar grouping approach [3]. Early research efforts, e.g. Gao et al. [1] proposed a fusion strategy for array contraction based on graph cutting, i.e. max-flow min-cut algorithm. Their representation model is a loop dependence graph that represents a single-entry single-exit region of the program, where all the edges have the same unit weight for partitioning. Kennedy et al. [6] proposed two fusion algorithms target both data locality and parallelism. They distinguish two types of loops, parallel and sequential, which is important for minimizing loop synchronization. Singhai et al. [8] studied a more restricted case of the loop fusion by considering locality with register pressure. This posed a leap forward to start considering resource usage.

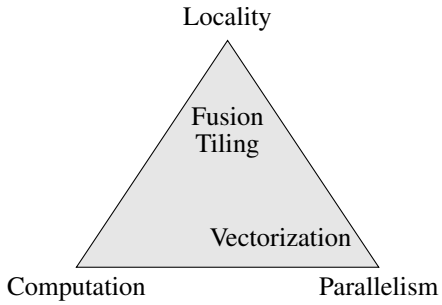


Figure 1: Optimization tradeoff.

Our work is inspired by those traditional loop fusion techniques yet differs in significant ways. First, kernel fusion targets locality improvement at the system level and emphasizes on inter-kernel locality improvement. Applications such as image processing consist of deep pipelines that communicate intensively with high-resolution images. This exhibits an opportunity for locality improvement. Second, we combine techniques from graph theory with domain-specific knowledge, which enables us to quantitatively and accurately estimate the degree of

[1] Artifact available at:https://doi.org/10.5281/zenodo.2240193

locality improvement. Consequently, the previously mentioned tradeoff can be explored more efficiently. Kernel fusion has been investigated in a number of research efforts [9] [10] [11] [12]. To the best of our knowledge, despite the close relationship between loop fusion and kernel fusion, this work is the first to solve the kernel fusion problem by combining a graph partitioning technique used in loop fusion with domain-specific and architecture knowledge. In previous work [12], Qiao et al. presented a basic kernel fusion technique in the context of a source-to-source compiler. This contribution proves kernel fusion an effective technique for locality optimization. In this paper, we extend their work in significant ways. Our contributions are as follows:

1) A formal description of the kernel fusion problem. Inspired by the loop fusion techniques, we define an objective function that represents an overall fusion benefit in terms of the execution cycles being saved. The goal of fusion is to maximize the benefit based on data dependence, resource usage, and kernel header information.
2) An analytic model for benefit estimation. The model combines domain-specific knowledge with architecture information to estimate potential locality improvement. Our model is able to explore the tradeoff between locality and redundant computation.
3) An algorithm to maximize the fusion benefit by employing a weighted minimum cut technique to search fusible kernels recursively. We demonstrate the steps using an image processing application.
4) Implementation of the proposed approach in an image processing DSL and source-to-source compiler framework called Heterogeneous Image Processing Acceleration (Hipacc)[2]. We also propose a technique for fusing stencil-based kernels with automatic border handling.

The remainder of this paper is organized as follows: Section II presents a formal description of the kernel fusion problem. An objective function is defined to represent the amount of benefit can be received by applying kernel fusion. To guarantee the legality of fusion, data dependence and resource constraints are examined. Then, the benefit estimation model is discussed in detail. Section III proposes an algorithm to solve the fusion problem. We use a real-world application to illustrate the steps of the proposed solution. Section IV describes our implementation environment. We show how kernels are fused in the context of an image processing DSL. Section V presents the evaluation outcome and we conclude our work in Section VI.

II. The Fusion Problem

Kernel fusion is a transformation technique that aggregates multiple fusible kernels into one. Target applications are represented in the form of directed acyclic graph (DAG) $G = (V, E)$. V is the set of vertices and E is the set of edges. Vertices in the graph represent kernels in the algorithm and edges represent the data dependence among kernels. If an edge connects two kernels $(v_i, v_j) \in E$, this indicates kernel v_j consumes the output produced by kernel v_i. A partition block P of G is a subset of G, where all vertices in P are connected. If both the source and destination vertex of an edge are in P, we say the edge is in P. Moreover, each edge $e \in E$ has a weight w_e, which is a positive number and represents the benefit gained by fusing the source and destination kernel of edge e. The weight of a partition block P is the sum of all the edge weights in P. In the remainder of this section, we first present the objective function and then discuss the two building blocks in detail, namely the legality of partition and the benefit estimation model.

A. Problem Statement

Given $G = (V, E)$ as input, with $w_e : E \mapsto \mathbb{R}_{>0}$ assigned to each edge $e \in E$ as weight, find a partition S with a set of partition blocks $S = \{P_1, ..., P_k\}$, $P_i = (V_i, E_i)$, $1 \leq i \leq k$ and $V_i \subseteq V, E_i \subseteq E$ where:

- All the partition blocks in S are *legal*. A partition block is legal if all the kernels inside can be fused to one, with data dependence being preserved and resource constraints being satisfied. (Discussed in Section II-B)
- S is pairwise disjoint, namely any two partition blocks in S do not share vertices, i.e., $\forall_{i \neq j}, V_i \cap V_j = \emptyset$.
- S covers G, namely the union of all the partition blocks in S equals to G, i.e., $V_1 \cup V_2 \cup ... \cup V_k = V$.

such that Eq. (1) is maximized.

$$\beta = \sum_{i=1}^{k} w_{P_i} \tag{1}$$

where w_{P_i} denotes the weight of a partition block $P_i = (V_i, E_i)$, i.e., $w_{P_i} = \sum_{e \in E_i} w_e$.

Eq. (1) is our objective function and β represents the benefit we receive by applying kernel fusion to the input application G, which represents the number of execution cycles being saved. Each partition block is one transformation, namely, all the kernels in that partition block will be fused into one kernel. The benefit we gain from this transformation is the weight of the block. The goal is to maximize the total benefit of all the obtained partition blocks. For example, if G is a partition block and all the kernels can be fused into one, we receive the maximum benefit in G. However, this situation rarely happens due to the dependence and resource constraints, which are discussed in the following subsection.

B. Legality

Image processing pipelines typically consist of kernels that take one or more images as input and produce one image as output. We define a kernel as a basic block that is a collection of statements, which can only be entered from the top and be left at the bottom. Fusing kernels in a partition block should preserve the original order of execution in G.

Listing 1 illustrates a simple fusion of three CUDA kernels. Assume the output of the source kernel is used by the intermediate kernel, and the output of the intermediate kernel is used by the destination kernel. Fusion will merge those three kernels into one, as depicted in Listing 1b. Two observations can be made in this example: First, fusion preserves the original

[2]http://hipacc-lang.org

execution order. After fusion, the new kernel executes three fusible kernel bodies in the original sequence. Second, fusion eliminates the data for inter-kernel communications, namely `OutSrc`, `OutIntmd`, `InIntmd`, and `InDest` in Listing 1a. Only the input of the source kernel and the output of the destination kernel are preserved.

For any possible partition block in G, assume it consists of a source kernel k_s, a destination kernel k_d, and one or more intermediate kernels. We identify four scenarios to discuss the data dependence, as depicted in Figure 2. Fusing a partition block is legal if the fused kernel body has no external dependence. Figure 2a shows a true dependence among kernels in the partition block, which is legal and straight-forward to fuse, similar to the example in Listing 1. Figure 2b shows another legal scenario that has not been explored in the previous work [12], where the inputs of k_s are shared by other kernels in the partition block. Fusion becomes illegal whenever an external dependence is introduced within the block, as depicted in Figure 2c and Figure 2d. As can be seen from Listing 1, only the input of the source kernel and the output the destination kernel are preserved after fusion. Therefore, any attempt to read or write another external memory location will introduce fusion preventing dependence. Hence, a partition block is legal to fuse only if no external dependence are introduced.

1) Resource Usage: Kernel fusion aims at locality improvement, which eliminates inter-kernel communication data from the slow global memory and brings it to the fast on-chip memories such as registers or shared memory[3] on GPUs. Those resources are shared among all the parallel computing units. Kernel fusion potentially increases the amount of resource usage, which might costs parallelism. If many kernels with high memory usage are fused into one, the loss of parallelism might outweighs the benefit of locality improvement.

In this work, we consider resource usage as another important constraint for legality. Good performance is achieved on GPUs

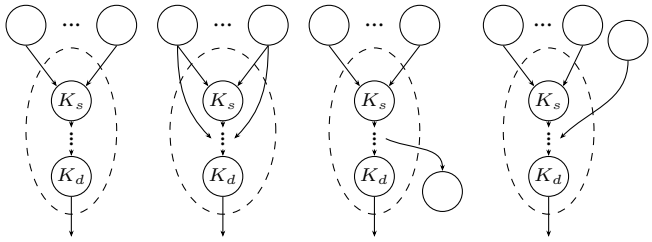
```
__global__ void KernelSrc (float *OutSrc, float *InSrc){
  ... = InSrc;
  // source kernel body ...
  OutSrc = ...;
}
__global__ void KernelIntmd (float *OutIntmd, float *InIntmd){
  ... = InIntmd;
  // intermediate kernel body ...
  OutIntmd = ...;
}
__global__ void KernelDest (float *OutDest, float *InDest){
  ... = InDest;
  // destination kernel body ...
  OutDest = ...;
}
```

(a) Fusible kernels

```
__global__ void KernelFused (float *OutDest, float *InSrc){
  ... = InSrc;
  // source kernel body ...
  // intermediate kernel body ..
  // destination kernel body ...
  OutDest = ...;
}
```

(b) Fused kernel

Listing 1: Simple kernel fusion example.



(a) True (b) Input (c) External output (d) External input

Figure 2: Dependence scenarios.

by latency hiding combined with massive parallelism. The number of thread blocks[3] that can run concurrently is limited by resource usage of the kernel, namely register and shared memory. At first glance, register pressure seems to be critical since the fused kernel has a larger body. Nevertheless, despite the increased size, the new kernel body is concatenated from the fusible kernel bodies. There exists no data dependence carried among them in the original code, except the input and output. We did not observe any increase in register pressure during kernel fusion. In comparison, the shared memory is more critical. Given a partition block P consists of a set of kernels $\{v_1, ..., v_n\}$, among which at least one kernel uses shared memory, and fusion tends to merge those n kernels into one kernel v_P. Let $f_{\mathrm{M_{shared}}}(v)$ represents the amount of shared memory used by kernel v in bytes. We define a partition block P as legal if Eq. (2) is satisfied for all the kernels in P that use shared memory.

$$\frac{f_{\mathrm{M_{shared}}}(v_P)}{\max(\{f_{\mathrm{M_{shared}}}(v) : v = v_i, ..., v_j\})} \leq c_{\mathrm{M_{shared}}} \qquad (2)$$

In Eq. (2), $\{v_i, ..., v_j\}$ is the set of kernels in P that use shared memory. $c_{\mathrm{M_{shared}}}$ is a user-given threshold to guarantee the occupancy [3] of the fused kernel. Eq. (2) prevents any dramatic increase of the shared memory usage due to kernel fusion.

2) Header Compatibility: Analogous to loop fusion, where the number of iterations of loops is examined to guarantee the feasibility, kernel fusion also checks the fusible kernels to have the same iteration space size. As introduced in previous work [12], all fusible kernels should have a compatible access granularity. We did not find this requirement too restrictive, since image processing pipelines typically operate on constant-size images. Therefore, we define a partition block P as legal if all the kernels have compatible headers, namely the same iteration space size and access granularity.

C. Benefit Estimation

For each edge $e \in E$ in G, we assign a weight w_e that represents the benefit gained by fusing the source and destination kernel of this edge. This can only be accurately estimated with the help of domain-specific knowledge as well as architecture

[3]CUDA terminology is used throughout this work. Shared Memory in CUDA is equivalent to the local memory in OpenCL, which is shared between all threads of a thread block. The amount available is architecture dependent. The kernel shared memory usage affects the number of blocks that can be active at once on a streaming multiprocessor (SM), which is a determining factor for occupancy. A high occupancy indicates a good utilization of resources and generally indicate a good implementation.

knowledge. For example on a GPU, we estimate the number of cycles being saved by relocating an intermediate image from global memory to shared memory or registers. After the relocation, we need to guarantee the fused kernel memory reading and writing from the desired location. One of the reasons that makes kernel fusion challenging on GPUs is that all the memory management tasks must be handled explicitly, which requires the programmer to know the behavior as well as the compute pattern in the target domain. In the remainder of this subsection, we first introduce the compute pattern and the hardware model proposed in this work. Then, we present formulas for benefit calculation.

1) Compute Pattern: One of the methods to categorize compute patterns in image processing is based on what information is used to map one image to another [13]. To compute each output pixel, if (a) one pixel is required as input, it is an element-wise operation, namely a point operator. Example functions are global offset correction, gamma expansion, tone mapping, etc. (b) a region of pixels is required as input, it is a stencil-based operation, namely a local operator. Example functions are Gaussian filter, convolution filter, median filter, etc. (c) one or more images are required as input, it is a reduction operation, namely a global operator. Example functions are histogram estimation, peak detection, etc. Point operators are the most straight-forward to be mapped on GPUs. Each pixel is computed by one thread and typically requires no shared memory usage. Local operators are more costly to compute, and each thread requires multiple pixels as input. Since each pixel is accessed multiple times, shared memory is generally employed to reduce the traffic to global memory. In this work, we consider point and local operators as the compute patterns for our target kernels. Together, they cover a wide range of image processing algorithms such as in computer vision and medical imaging.

2) Hardware Model: We consider a simplified memory model for GPUs that consists of registers, shared memory, and global memory. Global memory is the slowest to access and used by default. Assume t_g is the expected number of cycles to access a pixel from global memory, t_s is the expected number of cycles to access a pixel from shared memory, $IS(i)$ represents the iteration space size of an image i. The locality improvement can be defined as:

$$\delta_{\mathrm{M_{shared}}}(i) = IS(i) \cdot \frac{t_g}{t_s} \tag{3}$$

$$\delta_{reg}(i) = IS(i) \cdot t_g \tag{4}$$

where $\delta_{\mathrm{M_{shared}}}(i)$ represents the locality improvement by moving an image i from global memory to shared memory. $\delta_{reg}(i)$ represents the locality improvement by moving an image i from global memory to registers. Cost of global memory access on a GPU depends on many conditions such as access patterns, and can be hidden to some extent by switching threads. In this work, we use the global memory access latency (typically 400–800 cycles) to perform a conservative estimation. Cost of shared memory access is much cheaper compared with global memory, typically in a few cycles. However, it is still slower than registers, which can be accessed in a single cycle. Those variables are flexible and can be adapted for new architectures.

3) Fusion Scenarios: For an edge $e \in E$, assume $(k_s, k_d) \in E$, where k_s is the source and k_d is the destination kernel connected by this edge. k_s has n input images as $\{i_{s1}, ..., i_{sn}\}$, $n \geq 1$. i_e is the image that is produced by k_s and consumed by k_d. Relocating i_e is the goal of kernel fusion. Assume the benefit of fusing k_s and k_d is w, we distinguish four scenarios as follows:

Illegal: When k_s and k_d are illegal to fuse due to an external dependence or resource constraint, we do not assign benefit improvement. Instead of leaving w empty, we define an arbitrarily small positive number ε. The proposed algorithm requires all edges having a positive weight, which is discussed in the next section. In this scenario, the fusion improvement is simply given as $w = \varepsilon$.

Point-based: When k_s and k_d are legal to fuse, and k_d is a point kernel, we refer to this scenario as a point-based fusion. As previously introduced, a point kernel requires only one pixel for each output. On GPUs, this pixel can be computed by the same thread from the producer kernel k_s, regardless of the compute pattern. This is guaranteed by the granularity requirement in the header check. Therefore, each pixel can be kept in the register of each thread. In this scenario, we receive the best possible improvement by moving the intermediate image from global memory to register. The fusion improvement is given as:

$$w = \delta_{reg}(i_e) \tag{5}$$

Point-to-local: Kernel k_s and k_d are legal to fuse, k_s is a point kernel, and k_d is a local kernel. Whenever a local kernel is used as the consumer kernel, we need to explore the tradeoff between locality and redundant computation. Because on GPUs, the pixels required by the local kernel k_d are computed by different threads from the producer kernel k_s. As a remedy, each required pixel is recomputed in k_d, essentially trading the cheap computation for the costly memory access. In addition to the locality improvement δ, we introduce another term ϕ as the redundant computation cost of the fused kernel. Assume $cost_{op}$ represents the arithmetic computation cost of k_s given as:

$$cost_{op} = c_{ALU} \cdot n_{ALU} + c_{SFU} \cdot n_{SFU} \tag{6}$$

Where c_{ALU} represents the average cost (in cycles) of the arithmetic logic units (ALUs) operations, such as addition. n_{ALU} is the estimated number of ALU operations for the producer kernel k_s, c_{SFU} represents the average cost of the special function units (SFUs) operations, such as transcendental functions. n_{SFU} is the estimated number of SFU operations. Furthermore, we define IS_{k_s} as the sum of the iteration space size of all the input images for k_s. $sz(k_d)$ as the convolution mask size of the consumer kernel k_d, for example, 3×3. Thus the redundant computation cost ϕ is given as:

$$\phi = cost_{op} \cdot IS_{k_s} \cdot sz(k_d) \tag{7}$$

The redundant computation is performed to trade locality at the register level. Therefore, the locality improvement in this scenario is the same as in the previous scenario. The combined benefit of this scenario is given as:

$$w = \delta_{reg}(i_e) - \phi \tag{8}$$

As shown in Eq. (8), an expensive producer kernel that has many costly operations or loads many input images will increase the computation cost ϕ. Subsequently, the total benefit will be reduced. If the producer kernel is too expensive to compute, the cost ϕ might outweighs the locality improvement. In this case, the fusion should not be performed. This tradeoff has not been explored by previous work [12].

Local-to-local: Kernel k_s and k_d are both local and legal to fuse. Since local kernels use shared memory to reduce global memory communications. The consumer kernel k_d will read input from the shared memory. Therefore, the locality improvement is from global to shared memory, namely $\delta_{\mathrm{M_{shared}}}(i_e)$, which is already less than in the other scenarios. Regarding to the computation, the definition of $cost_{op}$ and IS_{k_s} remain the same as the point-to-local scenario. The convolution size $sz()$ must be refined. For example, a 3×3 local operator requires a window of 9 pixels as input, and 9 computations are required to produce one pixel. Nevertheless, if two 3×3 local kernels are fused, a window of 5×5 is required to produce one pixel. For the sake of simplicity, we assume square-shaped convolution mask sizes such as 3×3 or 5×5. The local-to-local fusion algorithm itself will be discussed in detail in our implementation section. Here, we provide the mathematical description only (for square regions):

$$g(sz(k_s), sz(k_d)) = \left(\sqrt{sz(k_d)} + \left\lfloor (\sqrt{sz(k_s)}/2) \right\rfloor \cdot 2\right)^2 \quad (9)$$

In Eq. (9), g represents the convolution size of the fused kernel. $sz(k_s), sz(k_d)$ denote the original convolution size of the source and destination kernels, respectively. For example, fusing a 3×3 source kernel with a 5×5 destination kernel yields a convolution size of 7×7 for the fused kernel. In this scenario, the computation cost ϕ is defined as:

$$\phi = cost_{op} \cdot IS_{k_s} \cdot g(sz(k_s), sz(k_d)) \quad (10)$$

and the total fusion benefit for this scenario is defined as:

$$w = \delta_{\mathrm{M_{shared}}}(i_e) - \phi \quad (11)$$

4) Putting it all together: We just discussed how w is computed, which represents the benefit of fusing two kernels in terms of the expected execution cycles being saved. Additionally, kernel fusion introduces other benefits. For example, reducing kernel launch overhead, enlarging the scope for further optimizations such as common sub-expression elimination. In the following, we define an independent term γ to give an overall estimation of these additional gains. We combine all the benefits, and clamp the value by setting a minimum number ε, which is also the benefit assigned to illegal fusions, to guarantee all the weights are positive. If any fusion indicates a benefit equals or smaller than zero, we should not fuse the kernels and can treat them as illegal scenarios. Finally, Eq. (12) defines the final value w_e assigned to each edge e:

$$w_e = \max(w + \gamma, \varepsilon) \quad (12)$$

After computing the weight of each edge, we can start applying our algorithm to maximize the total fusion benefit.

III. Algorithm

To maximize Eq. (1), we need to find the partition S, which is a set of partition blocks that maximizes the overall weight.

A. A Solution Based on Minimum Cut

Maximizing the weights inside all the partition blocks equals to minimizing the weights connecting the partition blocks. Assume G is partitioned into two blocks P_S and P_T. All the edges in G can have three possible locations: (a) Within block P_S. (b) Within block P_T. (c) Connecting P_S and P_T. We assume the total weight of all the edges in G is w_G, the weight of P_S and P_T is w_{P_S} and w_{P_T}, respectively. Let w_C denotes the sum of weights of edges crossing P_S and P_T. Then, we have the following equation:

$$w_G = w_{P_S} + w_{P_T} + w_C \quad (13)$$

Since the edges in G are fixed, the total weight w_G is a constant. Therefore, maximizing $w_{P_S} + w_{P_T}$ equals to minimizing w_C, which is a set of connecting edges that have a minimum total weight. This can be seen as a weighted minimum cut problem, and the set of edges can be found by applying a minimum cut algorithm. There exist extensive research efforts in graph theory on the minimum cut problem, including deterministic and randomized algorithms. We choose a well-known algorithm called the Stoer-Wagner algorithm to compute the minimum cut for our graph [14]. It is proposed for finding the minimum cut of an undirected edge-weighted graph, which is also applicable to directed graph as in our case. The algorithm is deterministic, efficient, and has a simple proof of correctness.

Our proposed algorithm works as follows: Given G, we assign a weight to each edge in G. The edge weight is computed by the benefit estimation model. Then, we initialize a ready set S_r and a working set S_p. S_r will be filled in by legal partition blocks and S_p contains all generated blocks during fusion. S_p is initialized with G as one single partition block. In each iteration, we inspect all partition blocks in S_p. If a partition block is legal or it contains only one kernel, it is put into the ready set S_r. If a partition block is illegal, we further divide it into two smaller partition blocks by cutting the block along its minimum edges. If there exist multiple sets of edges that have the same weight, the algorithm selects the first one encountered. The new partition blocks are added to the working set S_p. The steps are performed recursively, until the working set becomes empty and all the kernels are in the ready set. Finally, each partition block in the ready set is fused into one kernel. Obviously, for partition blocks consisting of a single kernel, no transformation has to be applied. The algorithm is listed in Algorithm 1.

B. Example

We use the Harris corner detector [15] to illustrate the proposed algorithm, as depicted in Figure 3. The application consists of nine kernels: $\{dx, dy\}$ are local operators that compute the derivation of an input image in x- and y-direction. $\{sx, sy, sxy\}$ are point operators that compute the square of the image. $\{gx, gy, gxy\}$ are local operators that approximate the Gaussian convolution of the image. Finally, $\{hc\}$ is a point

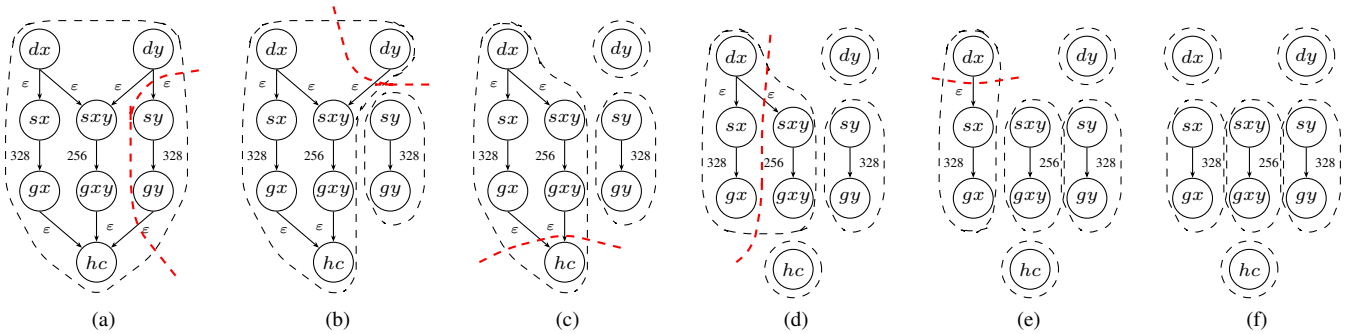


Figure 3: Kernel fusion algorithm applied to the Harris corner detector.

operator that measures the corner response of the image. Those nine kernels are connected by ten edges. Assume a constant-size image throughout the pipeline.

Step one is weight computation and assignment, as shown in lines 2-4 in Algorithm 1: Using our benefit estimation model, each edge is checked for its fusibility and categorized into one of the previously mentioned scenarios. Three edges are identified as legal: $\{(sx, gx), (sxy, gxy), (sy, gy)\}$. The other seven edges are identified as illegal. For example, $\{(dx, sx)\}$ cannot be fused due to an external output dependence exists. $\{(gx, hc)\}$ is illegal to fuse due to two external input dependence in kernel $\{hc\}$. Therefore, those edges are assigned the minimum weight ε. Next, we categorize the three legal edges as point-to-local fusion, based on the compute pattern of the kernels. We assume t_g is 400 cycles, c_{ALU} is 4 cycles, n_{ALU} is 2 for $\{sx, sxy, sy\}$, $sz()$ is 9 for $\{gx, gxy, gy\}$. Note that IS is not important here due to the constant-size image. IS can be simply replaced by the number of images for input. Furthermore, we omit the insignificant term γ for simplicity. Finally, we insert these numbers into Eq. (4), Eq. (6), Eq. (7), Eq. (8), and Eq. (12). We obtain and assign 328 to edge $\{(sx, gx), (sy, gy)\}$ and 256 to $\{(sxy, gxy)\}$, as depicted in Figure 3.

Algorithm 1: A recursive algorithm for kernel fusion

```
1  function GetLegalPartitions(G)
2      forall e ∈ E do
3          w_e ← EstimateBenefit(e) // Assign
              weight
4      end
5      S_r ← ∅ // ready set
6      S_p ← {G} // working set
7      repeat
8          forall p ∈ S_p do
9              if (|p| == 1) ∥ IsLegal(p) then
                   // single kernel or legal
10                 S_r ← S_r ∪ {p}
11                 S_p ← S_p \ {p}
12             else
                   // partition is illegal
13                 {p_1, p_2} ← MinCut(p)
14                 S_p ← S_p ∪ {p_1, p_2}
15             end
16         end
17     until S_p = ∅
18     return S_r
19 end
```

Step two is partitioning: We start by initializing a ready set S_r, a working set S_p, and assigning the DAG to S_p, as shown in lines 5-6 in Algorithm 1. In the first iteration, the legality of fusing the whole DAG is examined. No external dependence are detected but the resource usage does not satisfy Eq. (2), since $\{gx, gy, gxy, dx, dy\}$ are local kernels that use shared memory. Assume they all employ a convolution mask of size 3×3, based on our previous analysis, fusing two such kernels requires a convolution mask of size 5×5. There are three consumer kernels gx, gy, and gxy, which triples the memory usage. In total, the memory consumption increases five times if all those kernels would be fused to one. We limit $c_{\mathrm{M_{shared}}}$ to 2 in order to obtain high resource utilization. Therefore, the partition block is defined illegal and a min-cut of the graph must be found. As shown in lines 13-14 in Algorithm 1, we find the min-cut of the graph using the Stoer-Wagner algorithm. dx is used as the required starting vertex. We omit the detail of graph execution. After eight minimum-cut-phases of the algorithm, we obtain a global minimum cut of weight $2 \cdot \varepsilon$, as depicted in Figure 3a. The two generated partition blocks are put into the working set S_p. In the second iteration, the two blocks in S_p are examined. The smaller block $\{sy, gy\}$ is identified as legal and is put into the ready set S_r. The other block has an external output dependence from dy to sy, hence it is illegal and a min-cut needs to be found. The result of this iteration is shown in Figure 3b. Next, the steps are repeated until S_p becomes empty. Those iterations are depicted in Figure 3c, Figure 3d, Figure 3e, and Figure 3f.

C. Complexity and Optimality

Given a dependence graph $G = (V, E)$, the weight calculation step visits all the edges. Computing each edge weight takes constant time. Hence the whole weight computation has a running time $\mathcal{O}(|E|)$. The second step recursively cuts the partition blocks into smaller parts. Assume a worst-case as follows: No kernel in the graph can be fused. To get each kernel into the ready set, the graph needs to be cut until all the kernels are left in their own partition block. Moreover, each iteration only cuts one vertex from the partition block such that the partition blocks in the working set always have the

highest number of vertices possible. For this worst case, the Stoer-Wagner algorithm is applied in a maximum number of $|V|$ steps, within each step $|V|$ is reduced by one. The complexity of the Stoer-Wagner algorithm is $\mathcal{O}(|E'||V'| + |V'|^2 \log |V'|)$ for any graph $G' = (V', E')$ [14]. Given those conditions, we derive a worst-case running time of $\mathcal{O}(|E||V|^2 + |V|^2 \log(|V|!) + |E|)$ for Algorithm 1.

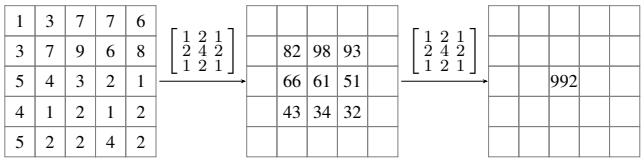
The output of our algorithm is a set of partition blocks, i.e. $\{P_1, ..., P_k\}$, each of which can be legally fused to one kernel and the sum of their connecting edges has minimum weight. The problem can be seen as a minimum weight k-cut problem. When k is undetermined, the problem has been proved NP-complete [16]. Hence, an exhaustive search is prohibited for applications with a large number of kernels. Moreover, compared with previous work [12], the proposed algorithm can explore fusion opportunities in a larger scope and detect beneficial scenarios that have been precluded. For example in Figure 3a, if all the kernels are point operators and no shared memory is used, the proposed algorithm would identify a legal fusion at the beginning and the whole graph would be fused into one kernel. Whereas in previous work, this opportunity would not be detected because only pair-wise fusion opportunities are considered and kernels are precluded as long as any constraint is met. The proposed technique in this work can explore more fusion opportunities, which is demonstrated in the evaluation section such as the fusion of the filter Unsharp.

IV. Implementation

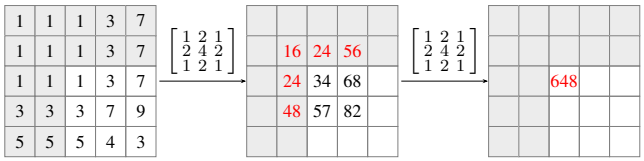
The proposed kernel fusion technique has been implemented in Hipacc, an image processing DSL and a source-to-source compiler [17] embedded in C++. Hipacc offers three operators based on what information contributes to an output, namely point, local, and global operators. Since our proposed technique is tailored to the point and local compute patterns, it can be fully automated in Hipacc. The compiler uses Clang AST as its intermediate representation. Kernel fusion has been implemented as an independent optimization pass within Hipacc. The framework supports a variety of backend targets ranging from multicore CPUs, GPUs to field programmable gate arrays (FPGAs) [18]. In previous work [12], fusion has been proposed only for point operator related fusion, namely point-to-point, local-to-point, and point-to-local. Those scenarios are relatively easy to implement because the correctness is guaranteed by the granularity. For example, to replace an input image with the output of a producer kernel, we only need to replace it with a register and inline the body of the producer kernel. In this paper, we extend the fusion technique to handle also automatic local-to-local fusion. We show that ensuring fusion correctness is challenging for this scenario, due to the exchange of pixels along the border between kernels. We propose an index exchange function to solve this problem.

A. Border Fusion

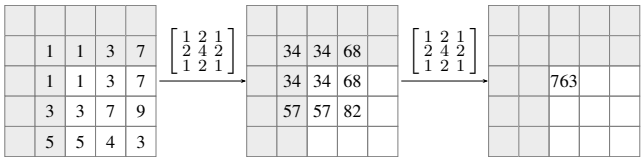
A local kernel in Hipacc takes a region of input pixels to produce one output pixel. Assume a producer kernel k_p and a consumer kernel k_c, both of which are local kernels and perform



(a) Body fusion: conv+conv



(b) Border fusion incorrect: clamp+conv+conv



(c) Border fusion correct: clamp+conv+clamp+conv

Figure 4: Local-to-local fusion.

a 3×3 convolution on an image. Before fusion, k_p requires 9 pixels to compute each intermediate pixel and k_c requires 9 intermediate pixels to compute one output pixel. To improve locality, all the required pixels can be loaded at once, hence trading redundant computations for locality as discussed in the previous section. We use a 2D Gaussian kernel and a randomly generated 2D integer matrix as an example depicted in Figure 4a. To compute one output pixel, the fused kernel requires 25 pixels input, as a 5×5 matrix. It is convolved with the Gaussian kernel to generate a 3×3 matrix. This intermediate data can be saved in the shared memory, where the accessing cost is reduced compared to the global memory. Finally, the 3×3 matrix is convolved with the Gaussian kernel to produce the output.

Nevertheless, this method is only applicable to the body of the processed image where all the accessed indices are valid, namely neither smaller than zero nor larger than the dimension size. Whenever the required pixels contain out-of-border indices, the iterative convolution method produces invalid results. Figure 4b shows the output value computed using the previous method at the top-left border position. In this section, we use Clamp as an example border handling mode. Nevertheless, the proposed technique also supports other modes such as mirror and repeat. First, images are padded based on the clamp mode and any out-of-border index is recomputed. The first pixel of the output image is computed by loading 25 pixels from the input. However, the result is wrong compared with the unfused implementation, as depicted in Figure 4c. In an unfused version, the input image is padded and convolved for the first kernel. After convolution, the intermediate data is stored back to the global memory. Then, it is loaded and padded again for the next kernel convolution. The padding for the second kernel is the reason for the incorrect result. The out-of-border pixels of the intermediate image should be recomputed before the second convolution starts.

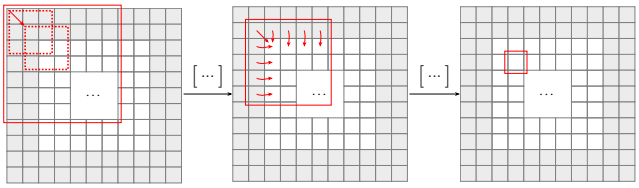


Figure 5: Index exchange for pixels in the halo region.

Whereas in Figure 4b, those pixels are unchanged throughout the computations.

B. Index Exchange

Border handling is required only for a small region of pixels in the whole image, depending on the mask size. We identify three regions for an image: (a) Interior region is the body of the image where no border handling is required, which is generally the largest region among the three and accounts for majority execution time. (b) Halo region is the border of the image where out-of-border accessing happens. The size of this region depends on the mask size. (c) Exterior region is not part of the image and is where padding applied. Figure 5 depicts those regions on three images. The outer gray region is the exterior region, the inner white square is the interior region, and the region in between is the halo region.

Two steps are required for local-to-local fusion to guarantee the correctness. First, the interior region of the fused kernel needs to be identified. Second, the computation for the halo region needs to be adapted. For simplicity, we assume square-shaped images and convolution masks. The width of the image is l_i, the width of the mask is l_k. Without fusion, the width of the interior region is $l_i - \lfloor (l_k/2) \rfloor \cdot 2$. The rest of the image is the halo region. For two kernels with mask size l_{k_p} and l_{k_c}, the width of the fused interior region becomes $l_i - \left\lfloor (\max(l_{k_p}, l_{k_c})/2) \right\rfloor \cdot 2$. We develop an index exchange method for local-to-local fusion. Figure 5 depicts an example. To compute the output pixel in the rightmost matrix, we need a 5×5 window from the middle matrix, assuming the convolution mask size of the first kernel is 3×3 and second kernel is 5×5. Each of those 25 pixels is examined with respect to its belonging region. If the pixel is in the interior region or the halo region, we do not change its index. If a pixel is in the exterior region, we exchange its index based on the border handling specified in the second kernel, as illustrated in the middle matrix in Figure 5, where Clamp mode is applied and the exterior pixels are exchanged with the border pixels. After the exchange, the corresponding access to the input image is also shifted accordingly. As can be seen in the leftmost matrix, where the 3×3 window required to compute the first exchanged pixel, is shifted one pixel to the right and one pixel to the bottom accordingly. Note that this step is only performed for the halo region of the output image, which is generally a small part of the computation.

We want to emphasize the importance of border handling, which we find being neglected in many existing works on fusion and DSL implementations. As can be seen from our analysis, the halo region grows quadratically with the number of local kernels being fused. Most work only considers the interior region after their optimization, and expect users to slice the resulting image. We believe that a correct border handling is a crucial ingredient for automating image processing code generation in a compiler.

V. Evaluation and Results

We analyze the speedups achievable from performing the proposed kernel fusion technique for six image processing applications on three Nvidia GPU targets. First, the experimental environment is introduced. Then, the benchmark applications are briefly described. Finally, we compare our implementation with the original version without kernel fusion and with implementations of previous work. Note that our main purpose here is to compare implementations using our proposed optimizations with non-optimized ones. For users who are interested in comparisons with other state-of-the-art, e.g. Halide, we refer to the work in [17].

A. Environment

The evaluation is based on Hipacc master branch version 50, which depends on Clang/LLVM 5.0. We optimize the CUDA code generation in Hipacc and employ NVCC for further compilation to the executable binary. The used CUDA driver version is 9.0 with NVCC release 8.0. Three GPUs are used in the evaluation: (a) Geforce GTX 745 facilitates 384 CUDA cores with a base clock of 1,033 MHz and 900 MHz memory clock. (b) Geforce GTX 680 has 1,536 CUDA cores with a base clock of 1,058 MHz and 3,004 MHz memory clock. (c) Tesla K20c has 2,496 CUDA cores with a base clock of 706 MHz and 2,600 MHz memory clock. For all three GPUs, the total amount of shared memory per block is 48 Kbytes, the total number of registers available per block is 65,536.

B. Applications

We chose six image processing applications to benchmark the kernel fusion technique: Sobel filter [19] as a well-known edge emphasizing operator applied in many edge detection algorithms. It uses two local operators to derive edge information along x- and y-direction, which are then combined to produce the gradient magnitude. The Harris corner detector [15] and the ShiTomasi good feature extractor [20] are classic filters for low-level feature extraction. Both algorithms involve the computation on a Hermitian matrix but interpret the Eigenvalues in different ways. The unsharp filter [21] is a technique for image sharpening, in contrast to its name. The implementation consists of a local kernel that blurs the image followed by three point kernels to amplify the high-frequency components. The night filter [22] [23] is a post-processing filter. It is implemented using three kernels that are linearly dependent and are executed in sequential order. Finally, we study an image enhancement algorithm [24] used in medical imaging for wireless capsule endoscopy. It uses geometric mean filter and gamma correction for de-noising and enhancement. Through the above mix of applications, we want to illustrate that our technique is widely applicable in image processing including multiple domains such as computer vision and medical imaging. Throughout the experiment, we use a constant-size image of 2,048 by 2,048

pixels. Note that an exception is the Night filter, which uses an image of 1,920 by 1,200 pixels in RGB format. The other five applications process in gray-scale.

C. Results

Figure 6 depicts the execution times in milliseconds. We implemented three versions for comparison: A baseline implementation. A basic kernel fusion implementation based on previous work [12]. And finally, an optimized implementation based on the technique proposed in this work. We performed 500 runs for each implementation on each GPU. To cover the uncertainties in the executions, we visualize the results in the box plot overlay shown in the graph. The whiskers cover the entire range of the obtained execution times including the minimum, the maximum, the 25 percentile, and the 75 percentile values. The box itself contains the intermediate 50% of all the results with a line in the middle represents the median. In most cases, the box is not visible due to the small variations in the obtained result.

Table I summarizes the achieved speedups for three comparisons: the optimized version over the baseline version, the basic version over the baseline version, and the optimized version over the basic version, respectively. As can be seen in Table I, the basic version from previous work [12] delivers the highest benefit on the algorithm Enhance, which is a chain of operations consisting of a local operator and two point operators. There are no external dependence and all the estimated benefit can be achieved. Nevertheless, the basic version failed in optimizing the filter Sobel and Unsharp. The filer Sobel consists of a local-to-local scenario and the filter Unsharp has shared input, both of which are rejected by the basic kernel fusion algorithm. As for the Harris and the ShiTomasi algorithms, the point-to-local scenarios are detected and the basic fusion is able to contribute speedups ranging from 1.04 to 1.28.

TABLE I: SPEEDUP COMPARISON.

	Optimized Fusion over Baseline					
	Harris	Sobel	Unsharp	ShiTomasi	Enhance	Night
GTX745	1.145	1.108	2.025	1.138	1.760	1.000
GTX680	1.344	1.377	3.438	1.357	1.920	1.020
K20c	1.146	1.048	2.304	1.149	1.809	1.000
	Basic Fusion over Baseline					
	Harris	Sobel	Unsharp	ShiTomasi	Enhance	Night
GTX745	1.044	1.002	1.007	1.046	1.413	1.001
GTX680	1.266	0.987	1.001	1.287	1.785	1.020
K20c	1.094	1.002	0.999	1.099	1.490	1.000
	Optimized Fusion over Basic Fusion					
	Harris	Sobel	Unsharp	ShiTomasi	Enhance	Night
GTX745	1.097	1.106	2.011	1.088	1.245	0.999
GTX680	1.061	1.394	3.435	1.055	1.076	1.000
K20c	1.047	1.046	2.304	1.046	1.214	1.000

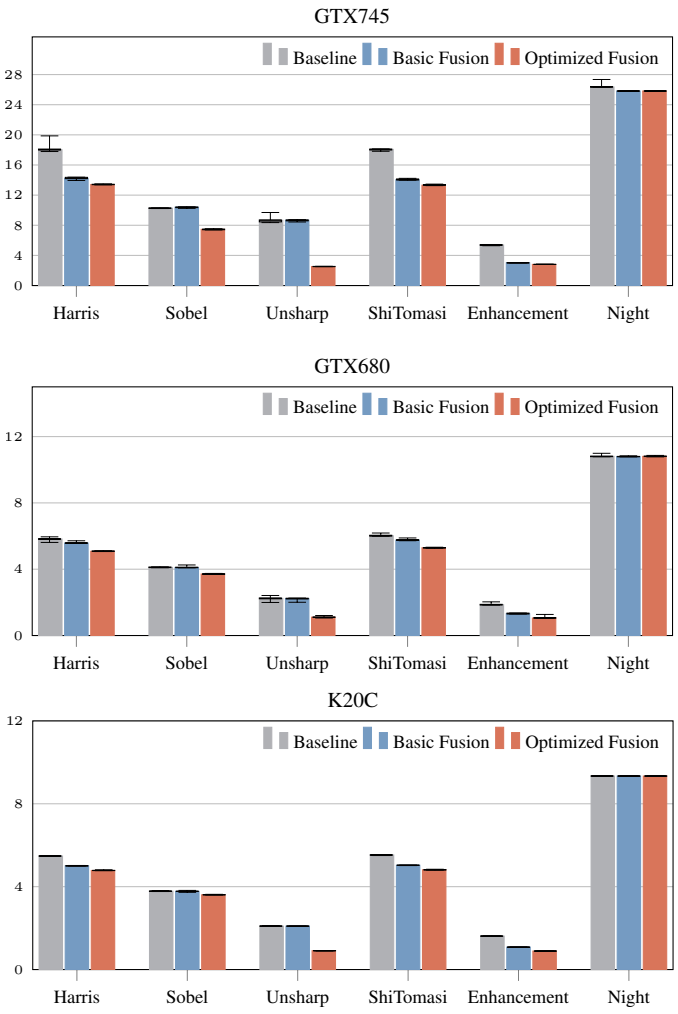


Figure 6: Execution times in ms.

The proposed optimized fusion technique has all the power of the basic fusion with larger fusion scopes and more optimizations. The filter Unsharp consists of four kernels. The shape of its DAG is similar to Figure 2b that all the four kernels require the source input image. Previous basic fusion will regard these dependence as external and invalid. In this work, we can detect this in a larger optimization scope and aggregate them into a single kernel. A significant speedup of up to 3.4 is achieved for this application. Furthermore, the Sobel operator computes both horizontal and vertical gradients, which is detected as local-to-local scenario and a speedup of up to 1.377 is achieved in the optimized fusion.

One observation from Table I is the optimization of the night filter. Among all the experimental results, only a maximum of 1.02 speedup is achieved, which is minor compared to other algorithms. The reason is the heavy computation cost of the kernels. Night filter consists of three kernels, `Atrous0`, `Atrous1`, and `Scoto`. The `atrous` (with holes) algorithm is applied two times (3×3, 5×5) to execute bilateral filtering. Those two local kernels are very expensive to compute. The number of ALU operations in the Hipacc implementation is 68. Using our benefit estimation model, the cost of redundant computation outweighs the locality improvement. Hence, the first two local kernels are not fused. The last kernel `Scoto` is a point operator, which applies a tone mapping curve that uses 89

TABLE II: GEOMETRIC MEAN OF SPEEDUPS ACROSS ALL GPUS.

	Harris	Sobel	Unsharp	ShiTomasi	Enhance	Night
Optm over Base	1.208	1.169	2.522	1.211	1.829	1.007
Basic over Base	1.131	1.000	1.002	1.139	1.555	1.007
Optm over Basic	1.068	1.173	2.516	1.063	1.176	1.000

ALU operations in the implementation. Kernel fusion detects a local-to-point scenario for `Atrous1` and `Scoto`, and fuses both into one kernel. The amount of speedup we receive from the fusion is much smaller compared to the computation cost of the kernels. Hence, compute-bound applications benefit less from kernel fusion. Nevertheless, most image processing algorithms consist of a complex pipeline of small operators rather than a few very complicated functions. Kernel fusion targets at algorithms expressed using common compute patterns, which are often memory-bound. Before conclusion, we provide an overview of all the speedups by computing the geometric mean across all three GPUs, as depicted in Table II.

VI. CONCLUSION

We presented an optimization technique called kernel fusion for data locality improvement. We proposed a technique for computing speedup benefits of kernel fusion based on the estimation of edge weights and formulating the problem as a minimum cut in a weighted graph. Based on this formalization, we proposed an algorithm to determine fusible kernels recursively and an analytic model to quantitatively estimate locality improvement by incorporating domain-specific knowledge and architecture details. The proposed technique has been implemented and integrated into a source-to-source compiler called Hipacc. We also presented a method for fusing stencil-based kernels with automatic border handling. Benchmark has been performed using six image processing applications on three GPUs. The results show that the proposed technique can achieve a geometric mean of speedups of up to 2.52. In the future, we want to extend our technique to other backend targets such as CPUs or FPGAs, and to explore further optimization techniques that can be used in conjunction with kernel fusion, such as kernel distribution.

APPENDIX

A. Artifact Abstract

This artifact describes the steps to reproduce the results for the CUDA code generation with kernel fusion in Hipacc[2] (an image processing DSL and source-to-source compiler embedded in C++), as presented in this paper. We provide the original binaries as well as the source code to regenerate the binaries, which can be executed on x86_64 Linux system with CUDA enabled GPUs. Furthermore, we include two python scripts to run the application and compute the statistics as depicted in Figure 6 in the paper.

B. Artifact Check-list

- Algorithm: Min-cut based kernel fusion in an image processing DSL.
- Program: Hipacc, Clang AST, CUDA.
- Compilation: Please check subsection C3.
- Transformations: Kernel fusion is implemented as an optimization pass in the Hipacc code generation.
- Binary: Included for x86_64 Linux (tested on Ubuntu 14.04 and 18.04).
- Run-time environment: Linux (Ubuntu 14.04 or later, 18.04 recommended), CUDA (9.0 or later, 10.0 recommended).
- Hardware: Please check subsection C2.
- Output: Execution time in milliseconds for each kernel.
- Experiments: Please check subsection E.
- Publicly available?: Yes.

C. Description

1) How Delivered: The pre-compiled binaries and the python scripts are available via:
https://www12.cs.fau.de/downloads/qiao/kernel_fusion/
The artifact source code is publicly available and hosted on GitHub at: https://github.com/hipacc/hipacc/tree/kernel_fusion

2) Hardware Dependencies: CUDA enabled GPUs are required. We used three Nvidia cards, as discussed in Section 5.1 in the paper:

- Geforce GTX 745 facilitates 384 CUDA cores with a base clock of 1,033 MHz and 900 MHz memory clock.
- Geforce GTX 680 has 1,536 CUDA cores with a base clock of 1,058 MHz and 3,004 MHz memory clock.
- Tesla K20c has 2,496 CUDA cores with a base clock of 706 MHz and 2,600 MHz memory clock.

For all three GPUs, the total amount of shared memory per block is 48 Kbytes, the total number of registers available per block is 65,536. GPUs with similar configurations are expected to generate comparable results.

3) Software Dependencies: To run the provided binaries, the prerequisites are the following:

- Clang/LLVM (5.0 or later), compiler_rt and libcxx for Linux (5.0 or later).
- CMake (3.4 or later), Git (2.7 or later).
- Nvidia CUDA Driver (9.0 or later).

To build Hipacc from source and re-generate the binaries, the prerequisites are the following:

- Clang/LLVM (6.0), compiler_rt and libcxx for Linux (6.0).
- CMake (3.4 or later), Git (2.7 or later).
- Nvidia CUDA Driver (9.0 or later).
- OpenCV for producing visual output in the samples.

For Ubuntu 18.04, most of the software dependencies can be installed from the repository. CUDA driver is available on Nvidia download website.

4) Datasets: The provided binaries generate random images of size 2,048 by 2,048 pixels, hence no additional data is required. Nevertheless, some real-world images are included in the repository, which can be used for visualization. Images of the same size are expected to have similar execution time. Throughout our experiment, we have used image size of 2,048 by 2,048 pixels. Note that an exception is the Night filter, which computes an image of size 1,920 by 1,200 pixels. The choice of image size was made randomly, and not relevant to the purpose of kernel fusion.

D. Installation

This subsection illustrates the installation steps to build Hipacc from source, the previous software dependencies should be met prior to the following steps.

1) Clone the Hipacc repository.

```
$ git clone https://github.com/hipacc/hipacc.git
$ cd hipacc/
```

2) Switch to kernel fusion branch and update the submodule samples.

```
$ git checkout kernel_fusion
$ git submodule init && git submodule update
```

3) Build Hipacc from source.

```
$ mkdir build && cd build
$ cmake ../ -DCMAKE_INSTALL_PREFIX="pwd"/release
$ make && make install
```

Hipacc should be successfully installed in the release folder.

E. Experiment Workflow

After a successful installation, we can go to the release folder and generate binaries.

1) Disable/Enable kernel fusion.

```
$ cd release/samples/common/config
disable fusion by (-fuse off) or
enable by (-fuse on) in cuda.conf.
```

2) Generate binary and run the application, e.g. Harris corner.

```
$ cd ../../3_Preprocessing/Harris_Corner/
$ make cuda
```

The above step should produce a binary *main_cuda*, which should be comparable to the provided binary. The steps can be repeated for all six applications in the paper, which are located within the preprocessing directory and the postprocessing directory respectively.

After having the binaries, the provided python scripts can be used to collect the total execution time and the statistics as shown in Figure 6 in the paper. Note the name of the executable binary might need to be adapted in the python script.

F. Evaluation and Expected Result

The above experiment workflow is expected to output the execution times comparable to Figure 6 in the paper. The values in Figure 6 are also available in text format via the provided download website.

The reproduced GPU execution results are expected to have small variations compared with the original presented data. For example, the median value among 50 runs of a baseline Harris corner implementation is expected to have a variation between 0.05 ms and 0.1 ms on a GTX680. In general, we expect a variation of +/- 0.1 in the final speedup comparison. The gains in Table 1 and Table 2 in the paper can be derived from the median value of the obtained statistics.

G. Notes

- From experience, the first call to a GPU device takes longer time. We did several dummy calls to the GPU at the beginning of our evaluation.
- The provided binaries were generated and documented at the beginning of the work, which might use an older version of Clang/LLVM, CUDA Driver/Runtime, and MVRTC version. Nevertheless, the results are expected to be comparable with executions with newer drivers.

ACKNOWLEDGMENTS

This work is partially supported by Siemens Healthineers and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Projektnummer 146371743 — TRR 89 “Invasive Computing.”

REFERENCES

[1] G. R. Gao, R. Olsen, V. Sarkar, and R. Thekkath. “Collective Loop Fusion for Array Contraction”. In: *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer, 1993, pp. 281–295.

[2] F. Irigoin and R. Triolet. “Supernode Partitioning”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. (San Diego, CA, USA). ACM, Jan. 1988, pp. 319–329. DOI: 10.1145/73560.73588.

[3] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. “Automatically Scheduling Halide Image Processing Pipelines”. In: *ACM Transactions on Graphics* 35.4 (July 2016), 83:1–83:11. DOI: 10.1145/2897824.2925952.

[4] A. Ashari, S. Tatikonda, M. Boehm, B. Reinwald, K. Campbell, J. Keenleyside, and P. Sadayappan. “On Optimizing Machine Learning Workloads via Kernel Fusion”. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP 2015. San Francisco, CA, USA: ACM, 2015, pp. 173–182. DOI: 10.1145/2688500.2688521.

[5] K. S. McKinley, S. Carr, and C.-W. Tseng. “Improving Data Locality with Loop Transformations”. In: *ACM Transactions on Programming Languages and Systems* 18.4 (July 1996), pp. 424–453. DOI: 10.1145/233561.233564.

[6] K. Kennedy and K. S. McKinley. “Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution”. In: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer, 1994, pp. 301–320.

[7] R. T. Mullapudi, V. Vasista, and U. Bondhugula. “PolyMage: Automatic Optimization for Image Processing Pipe-lines”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’15. Istanbul, Tur-key: ACM, 2015, pp. 429–443. DOI: 10.1145/2694344.2694364.

[8] S. K. Singhai and K. S. McKinley. “A Parametrized Loop Fusion Algorithm for Improving Parallelism and Cache Locality”. In: *Computer* 40.6 (1997), pp. 340–355. DOI: 10.1093/comjnl/40.6.340.

[9] G. Wang, Y. Lin, and W. Yi. “Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU”. In: *Proceedings of the 2010 IEEE/ACM International Conference on Green Computing and Communications and International Conference on Cyber, Physical and Social Computing (GREENCOM-CPSCOM)*. IEEE Computer Society, 2010, pp. 344–350. DOI: 10.1109/GreenCom-CPSCom.201 0.102.

[10] H. Wu, G. Diamos, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. “Optimizing Data Warehousing Applications for GPUs Using Kernel Fusion/Fission”. In: *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*. May 2012, pp. 2433–2442. DOI: 10.1109/IPDPSW.2012.300.

[11] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska. “Optimizing CUDA Code by Kernel Fusion: Application on BLAS”. In: *The Journal of Supercomputing* 71.10 (Oct. 2015), pp. 3934–3957. DOI: 10.1007/s11227-015-1483-z.

[12] B. Qiao, O. Reiche, F. Hannig, and J. Teich. “Automatic Kernel Fusion for Image Processing DSLs”. In: *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. (St. Goar, Germany). ACM, May 28–30, 2018, pp. 76–85. DOI: 10.1145/3207719.3207723.

[13] I. Bankman. *Handbook of Medical Image Processing and Analysis*. Jan. 2009.

[14] M. Stoer and F. Wagner. “A Simple Min-cut Algorithm”. In: *J. ACM* 44.4 (July 1997), pp. 585–591.

[15] C. Harris and M. Stephens. “A Combined Corner and Edge Detector”. In: *In Proceedings of the Fourth Alvey Vision Conference (AVC)*. (Manchester, UK). Sept. 1988, pp. 147–151.

[16] O. Goldschmidt and D. S. Hochbaum. “A Polynomial Algorithm for the K-cut Problem for Fixed K”. In: *Math. Oper. Res.* 19.1 (Feb. 1994), pp. 24–37. DOI: 10.1287/moor.19.1.24.

[17] 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* 27.1 (Jan. 2016), pp. 210–224. DOI: 10.1109/TPDS.2015.2394802.

[18] M. Özkan, O. Reiche, F. Hannig, and J. Teich. “FPGA-based Accelerator Design from a Domain-Specific Language”. In: *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*. (Lausanne, Sw- itzerland). IEEE, Aug. 29–Sept. 2, 2016. DOI: 10.1109/FPL.2016.7577357.

[19] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[20] J. Shi and C. Tomasi. “Good features to track”. In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. (Seattle, WA, USA). IEEE, June 21–23, 1994, pp. 593–600. DOI: 10.1109/CVPR.1994.323794.

[21] G. Ramponi. “A Cubic Unsharp Masking Technique for Contrast Enhancement”. In: *Signal Process.* 67.2 (June 1998), pp. 211–222.

[22] 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.

[23] 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.

[24] S. Suman, F. A. Hussin, A. S. Malik, N. Walter, K. L. Goh, I. Hilmi, and S. h. Ho. “Image Enhancement Using Geometric Mean Filter and Gamma Correction for WCE Images”. In: *Neural Information Processing*. Ed. by C. K. Loo, K. S. Yap, K. W. Wong, A. T. Beng Jin, and K. Huang. Cham: Springer International Publishing, 2014, pp. 276–283.