A Case Study in Implementation-Space Exploration

Alexander Lier Linus Franke Marc Stamminger Kai Selgrad

Computer Graphics Group, Friedrich-Alexander University Erlangen-Nuremberg, Germany

{alexander.lier, linus.franke, marc.stamminger, kai.selgrad}@fau.de

ABSTRACT

In this paper we show how a feature-oriented development methodology can be exploited to investigate a large set of possible implementations for a real-time rendering algorithm. We rely on previously published work to explore potential dimensions of the implementation space of an algorithm to be run on a graphics processing unit (GPU) using CUDA. The main contribution of our paper is to provide a clear example of the benefit to be gained from existing methods in a domain that only slowly moves toward higher level abstractions. Our method employs a generative approach and makes heavy use of Common Lisp-macros before the code is ultimately transformed to CUDA.

1. INTRODUCTION

When developing algorithms to be used in time-critical application domains, such as real-time rendering, many different implementation variants need to be evaluated to arrive at the method that best exploits the available resources. This is even more true in research where the spectrum of solutions to investigate is potentially larger. Higher-level languages such as Common Lisp, can ease this process of finding the best solution considerably. However, while providing great flexibility and productivity, they are not available in all domains and for all applications. This can be due to technical limitations (vendor-specific languages, compatibility) or policies (certified processes, coherent working environment) as well as due to resistance from developers unwilling to embrace change.

In earlier work we proposed formulating C and similar languages in an S-Expression syntax and only transforming them to their native notation as late in the process as possible [22] using C-Mera[1]. This allows employing a macro-heavy methodology to generate many different variants of an input program, and to express it on a very high level. We believe that this scheme, even if very unfamiliar to programmers used to C, can help make higher level paradigms available to that domain (also by supporting guerilla adoption [22, 21]).

For domain exploration we recently proposed employing a feature-oriented programming paradigm to implement the exploration process [21]. To this end we use cm-fop[2], a very lightweight library that provides this feature-oriented programming model to C-Mera.

In our case study we show how these methods can be applied to find highly efficient implementations of a post-processing depth of field effect for real-time rendering.

The following section, related work, is divided into two parts. At first we cover generative meta-programming techniques followed by rendering methods for depth of field. After a short review of C-Mera in a dedicated section, we depict several aspects and details of our depth of fild algorithm. Details and examples of the actual meta programming approach are given in the implementation section. Thereafter, the generated resulting code is discussed and evaluated, and our findings are summarized in a short conclusion.

2. RELATED WORK

In this paper we propose employing a generative methodology to explore the space of possible implementations of real-time depth of field rendering algorithms. Section 2.1 gives a short review of generative programming techniques while Section 2.2 provides an introduction to depth of field rendering. The latter is divided further, whereby the first part discusses depth of field in general. Afterwards, the difference between gathering and scattering methods is elucidated, followed by a description of how the scattering algorithm can be harnessed for GPU application.

2.1 Generative Meta-Programming

In the following we will focus on related work concerned with general purpose methods providing domain-specific abstractions. For a comprehensive summary of general generative programming methods see Czarnecki and Eisenecker [4].

The most ubiquitous approach to generative programming is C++ template meta programming (TMP) [26, 4]. It has been applied to a wide range of problems, also in graphics (for example with RTfact [23], a ray tracing library). We believe that, while certainly convenient and unobtrusive to use when working in a C++ environment, exploration is seriously impaired by this approach as the maintenance overhead of the meta code becomes a burden in itself [7, 13, 21]. Also note that TMP is only available in C++, only partially

[1] github.com/kiselgra/c-mera

[2] github.com/kiselgra/cm-fop

European Lisp Symposium 2016, Krakow, Poland

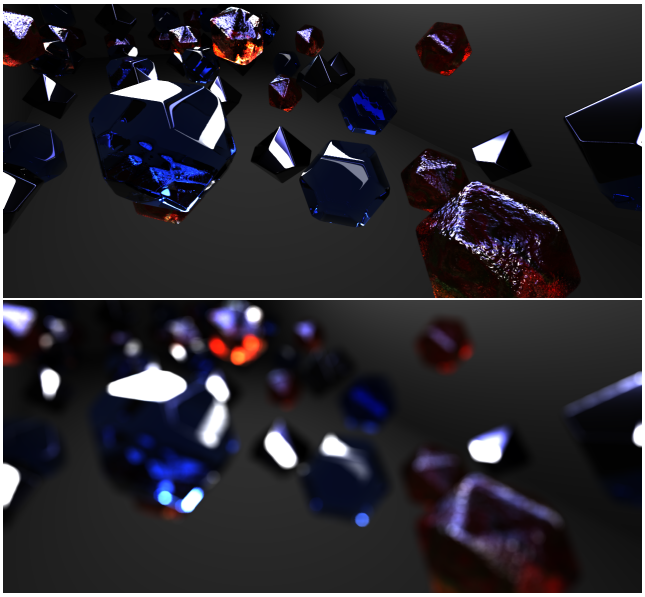

Figure 1: Rendered image (top) and post processed image showing depth of field (bottom).

in CUDA [15] and not in other, very similar languages such as OpenCL [8] and GLSL [11].

C-Mera is a multi-stage programming language [24], that is, a language that is embedded in a host language (in our case Common Lisp). Embedded languages are compiled into the host language and the resulting program is then further compiled or interpreted. In the case of C-Mera the evaluation of the input program constructs the internal Common Lisp representation that is then pretty printed to C-style code.

Examples of other multi-stage programming languages are MetaOCaml [3], an extension for staging OCaml, Terra [6], a low-level language embedded in Lua, and AnyDSL [14], which gives the programmer explicit control over when certain parts of the program are to be evaluated.

Feature-oriented programming [17, 2] offers additional degrees of freedom regarding programming versatility. A *feature* is a unit of functionality that provides an interface for configuration. A feature-oriented program is then composed of features that together provide its implementation. This scheme aims to provide well-structured programs that can be configured to provide different incarnations by varying the implementations of the underlying features [1]. Our work relies heavily on feature-oriented programming, as it is demonstrated in more detail in Section 5.

2.2 Depth of Field Rendering

Depth of field is a physical effect caused by the finite size of the lens in an imaging system (e.g. in a camera or in the eye). Rays of light are collected through the aperture of the system (the pupil of the eye) and focused by the lens on the image sensor (the retina). As the refractive power of the lens can only be in one state at any given time only light arriving from a given distance is actually in focus and mapped to a single point on the sensor. Light from a different distance is mapped to a circular region (or, in case of a camera, to a region in the shape of the camera's aperture, known from polygonal shapes of out-of-focus lights in movies and artistic photographs). This *circle of confusion* is the reason out-of-focus objects appear blurred. Figure 1 shows an example computed using our implementation of a post-processing method.

When synthesizing images this is an important effect for generating more plausible results, as the depth perception is skewed for scenes that are shown completely in-focus. Demers [5] describes this effect in great detail and provides an overview of methods of generation. In the following we will only touch on a few methods and refer to Demers [5] for more details.

Gathering vs Scattering. The most common method to implement depth of field is by adding it to an previously generated image. This can be done in two ways, namely via *gathering* or *scattering*. Gathering approaches apply a (bilateral) image-space blur filter where the filter's size depends on the current pixel's circle of confusion [5, 18]. The main limitation of these methods is that the ordering of the participating pixels is not considered correctly and only a weighted average is computed [19]. With scattering methods the problem is approached differently: the rendered image is interpreted as a point-sampled scene representation and the individual points are scaled to form circles according to the original pixel's circle of confusion. These scaled points ("splats") are then sorted and accumulated (in order) as semi-transparent objects, thus ensuring correct weighting [16, 12]. The limitation of these methods is that they require a global ordering of the splats be established, which is an expensive operation.

Recent work on tiled shading [9] and particle accumulation [25] can be applied to remove this limitation of scattering depth of field algorithms in a straightforward fashion [19].

Fast GPU Particle Accumulation. High-performance particle accumulation can be implemented by employing a tiling-based scheme that maps well to graphics hardware. Instead of globally sorting the particles, they are binned into screen-space tiles (e.g. of 16×16 pixels, potentially a $1 : n$ mapping) and the tiles are then sorted independently and in parallel [25]. The computation of the contribution for each pixel is then a simple process that traverses the list of particles in the pixel's associated tile. This process, too, maps very well to hardware as (in CUDA terms) the execution can be set up such that each thread-block traverses a single tile, and thus the threads in a warp run with great coherency.

In Section 4 we list choices of how to implement the accumulation phase of this technique for depth of field rendering. We also show how C-Mera, together with cm-fop, can be used to explore the space of possible solutions with feature-oriented programming.

3. BRIEF REVIEW OF C-Mera

C-Mera is a simple transcompiler embedded in Common Lisp. It allows writing programs in an S-Expression syntax that is transformed to C-style code providing simple extension for languages with similar syntax on top of the core C support. For example, the C-Mera distribution provides modules for C++, CUDA, GLSL and OpenCL. The main goal of providing an S-Expression syntax is to write the compiler such that it evaluates the syntax to construct a syn-

```
1  (for ((int i 0) (< i num-per-tile) ++i)
2    (decl ((dataSpl elem (aref lists tile-index i))
3           (int dist-x (- (funcall elem.x) gid.x))
4           (int dist-y (- (funcall elem.y) gid.y))
5           (float coc-sq (funcall elem.sq-coc)))
6      (if (<= (+ (* dist-x dist-x) (* dist-y dist-y)) coc-sq)
7        (decl ((float area (* coc-sq 3.1415f))
8               (float alpha (/ 1.0f area)))
9          (+= color.x (* alpha-dest alpha elem.r))
10         (+= color.y (* alpha-dest alpha elem.g))
11         (+= color.z (* alpha-dest alpha elem.b))
12         (*= alpha-dest (- 1.0f alpha))))))
```

```
1  for (int i = 0; i < num_per_tile; ++i){
2      dataSpl elem = lists[tile_index][i];
3      int dist_x = elem.x() - gid.x;
4      int dist_y = elem.y() - gid.y;
5      float coc_sq = elem.sq_coc();
6      if (((dist_x * dist_x) + (dist_y * dist_y)) <= coc_sq) {
7         float area = coc_sq * 3.1415f;
8         float alpha = 1.0f / area;
9         color.x += (alpha_dest * alpha * elem.r);
10        color.y += (alpha_dest * alpha * elem.g);
11        color.z += (alpha_dest * alpha * elem.b);
12        alpha_dest *= (1.0f - alpha);
13     }
14 }
```

Figure 2: The most basic particle accumulation loop as used as input for C-Mera (top) and the resulting generated C-style code (bottom).

tax tree when the input program is read, thereby allowing interoperability with the COMMON LISP-system, most importantly by providing support for Lisp-style macros. To keep this part short we refer to the original C-MERA paper [22] for a more detailed description of the system and its implementation.

With the use of macros the input program no longer represents a plain syntax tree, but a semantically annotated tree that is transformed according to the implementation of the semantic nodes (macros). The utility of such a system ranges from simple, ad-hoc abstractions and programmer-centric simplifications [22] to providing otherwise hard to achieve programming paradigms for C-like languages [21] and even to fully fledged domain specific languages [20]. In this paper we provide a case study of applying C-MERA to provide a higher level programming paradigm for GPU algorithm development.

Figure 2 shows an example from our application domain, which is processed by C-MERA, and C-style code that is generated by it. The overall appearances of the input (top) and the generated code (bottom) resemble each other but the former applies S-Expressions, whereas C-style syntax and infix expressions are required in the latter. As can be seen, C-MERA supports convenient and essential language elements from C, for example the member accessor *color.x* (line 10,11,12), infix increment *++i* (line 1, top), type suffix *3.1415f* (line 7, top), and further notations that are not shown here. Additionally, C-MERA renames variable names that are restricted in C, for example *dist-x* (line 6, top).

4. IMPLEMENTATION-SPACE

We focus our analysis on the accumulation of particles, which can be described briefly as follows: Every pixel of the resulting image must be synthesized form a number of particles that might affect it. To do so, the program has to iterate through a previously sorted list with possible candidates in reach that might affect the resulting pixel. Every tile, a group of 16×16 pixels, has one associated list, therefore a list is shared by 256 pixels.

The process of sorting the entries in the tile lists, as well as the subsequent accumulation benefit greatly from a compact memory layout in which single entries can be transferred en-bloc. For our CUDA implementation this means that the entries should be no more than 16 bytes, such that they can be encoded as `uint4`. However, the required fields are the pixel's (high resolution) color, screen-space position and camera distance. This data can be stored compactly, but the best choice not only depends on the number of bits reserved for each entry, but also on the effort to unpack the respective fields and how how likely they will be accessed. In our implementation we tested two different basic node layouts.

The most obvious point to evaluate different implementations is how the CUDA warp and block configuration is set up and employed during traversal. Here we evaluated two different approaches: Loading large chunks (256 elements) of the tile-lists (with an average length of 1300) to a buffer of shared memory, and then processing the list chunk-by-chunk with all threads working in parallel on the same data. This, however, requires synchronization after the data has been loaded to ensure that it is available to all threads. The second approach is to only load blocks of warp size (i.e. 32) to shared memory and process smaller chunks. The benefit of this approach is that no synchronization is necessary, however, at the cost of smaller batches in the shared cache and redundant loads on the block level.

To gain greater insight into the effect of blending front-to-back vs back-to-front, both approaches have been analyzed. Naturally, the front-to-back method performs better [25] as it offers the option to terminate early when the pixel is saturated. This leads to the question of how finely checking for early termination is advisable: after each accumulation step, or only after each chunk, which interacts with the aforementioned chunk size?

We also evaluated many small-scale optimizations such as explicitly enabling caching to L1, storing vs computing certain values and peeling off parts of loops to remove conditionals from the inner loop. In the end we arrived at 320 different (and meaningful) *combinations* that are implemented using CM-FOP in 300 lines of feature definitions, feature implementations and the algorithm-template that expand to more than 16000 lines of CUDA code.

5. IMPLEMENTATION

As the previous section shows, our implementation space expands into multiple dimensions; thus we must consider and evaluate many versions of the accumulation loop, which might differ heavily from each other and from the unoptimized version shown in Figure 2. Starting from the basic implementation and with a rough draft of the desired variations, we can incrementally extend the existing solution with functionalities, also in reaction to the results from previously tested variants. This leads to an iterative and especially explorative programming methodology.

In this section we will discuss our meta implementation, starting with an explanation of why we have chosen features over plain macros. The following examples will at first focus on feature usage and later examine target and feature interaction in more detail.

```
1  (defmacro early-out (target condition &body body)
2    (cond ((eql target 'no-early-out)
3           `(progn ,@body))
4          ((eql target 'blockwise-early-out)
5           `(if ,condition
6              (progn ,@body)
7              (break)))
8          (t
9            `(error "The target ~a is not specified" ',target))))
```

Figure 3: Example macro implementation with multiple expansion possibilities.

From Macros to Features. Relying solely on COMMON LISP's macro system to facilitate previously mentioned extensions has the risk of becoming tedious, since particular variation points can mutually exclude each other and macros do not support convenient configurability of their implementation. Therefore, if wanting to write a macro that combines multiple varying results, one has to implement each of its possible expansions inside a single block of conditionals. An example of such a macro is given in Figure 3. Therefore, we employ CM-FOP, C-MERA's library for feature-oriented programming to ease this procedure. This library enables using features that are essentially macros with a built-in system that automatically implements and resolves conditional expansion. A feature-oriented definition equivalent to the macro from Figure 3 is shown in Figure 4. In contrast to plain macros the feature system is able to recognize the desired expansion code by means of a *configuration variable*, which can be defined and stays valid within a lexical scope and thus supports nesting of different *configurations*. Additionally, the feature system decouples the feature definition from its implementations, with the result that the definition of a feature is only required once and its likely multiple and divergent implementations can be written individually. Furthermore, writing a feature implementation does not require to manually declare its dependencies, nor define conditional expansion. Thus, unlike a macro that incorporates such a behaviour, feature implementations only require minimal boiler-plate code. Nevertheless, it should be mentioned that CM-FOP's feature system relies on COMMON LISP's macro and object system, but without the need to manually define the conditional expansions, CM-FOP improves the handling of multiple implementations over plain macros considerably.

Feature Setup. As a first example we introduce a simple feature to the algorithm from Figure 2. The most basic part of extending the particle accumulation is to exit the loop when the pixel is opaque because collecting further elements will not change the resulting color. To do so we construct a feature equal to the macro shown in Figure 3 that wraps the body with an if-statement and uses a break operation to exit the loop. The corresponding feature setup is shown in Figure 4.

Before we implement features, we define their possible targets (lines 1 and 2). Features are defined once (line 4) and support one implementation per target combination (lines 6 to 12). Targets can be derived from each other, as it is the case here, thus the combination of the currently used most specific target and its available implementations determine the expanded code. For example, if the implementation for *blockwise-early-out* is not given, but that target is used, the

```
1  (define-target no-early-out)
2  (define-target blockwise-early-out no-early-out)
3
4  (define-feature early-out (condition &body body))
5
6  (implement early-out (no-early-out)
7    `(progn ,@body))
8
9  (implement early-out (blockwise-early-out)
10   `(if ,condition
11      (progn ,@body)
12      (break)))
```

Figure 4: Construction of a feature for a conditional break

more general *no-early-out* will be used. However, if implementations for more specific targets are given, but a less specific target is used, the best fitting implementation (according to CLOS's [10] method lookup) is chosen. As can be seen in Figure 4, the body for a feature implementation is similar to the body of a standard macro, but there are multiple implementations for the same feature. The upper implementation (line 6 and 7) returns the body passed in without modifications. The other one (line 9 to 12) splices the body inside an if-statement, places the condition in the designated position, and introduces a break-statement as the else case.

Elementary Feature Utilization. The example application shown in Figure 5 depicts the use of features (top) and their resulting code (bottom), whereby feature applications are highlighted in orange and targets in green. Multiple targets can be combined with *make-config* into one single configuration (top, line 1 and 6). The *with-config* form takes a configuration as an argument and declares it as locally valid for features used within its lexical scope. Depending to the configuration used, each *early-out* feature expands into different code.

The *no-early-out* target adds no further code nor changes the body, whereas the *blockwise-early-out* target adds an if-clause and a break-statement.

This behaviour is the essential element that we want to utilize. Normally if we introduce additional variations, we must clone and partially rewrite every version for every additional divergence. Thus, the number of possible implementations to write grows exponentially. Our solution for this problem, as we already proposed in previous work [21], is to use a single, general implementation that handles each diverging point individually by applying the proper feature expansion.

Based on the examples from Figure 4 and Figure 5 an applicable multi-variant-aware implementation is shown in Figure 6. As can be seen, we now only require one implementation of the accumulation-loop that can expand into multiple versions depending on the configuration passed in.

Pinpoint Implementation. By adding an ever increasing amount of expansion possibilities to the unified implementation, we arrive at the most general implementation of the accumulation loop, which is shown in Figure 7. This algorithm template is capable of expanding into 320 versions of the accumulation loop. Most of the features use (highlighted in orange) are implemented similarly to previous examples

```
1  (with-config (make-config no-early-out)
2    (for ((int i 0) (< i num-per-tile) ++i)
3         (early-out (> alpha-dest 0.01)
4          ....
5
6  (with-config (make-config blockwise-early-out)
7    (for ((int i 0) (< i num-per-tile) ++i)
8         (early-out (> alpha-dest 0.01)
9          ....
```

```
1  for (int i = 0; i < num_per_tile; ++i){
2      dataSpl elem = lists[tile_index][i];
3      ...
4  }
5
6  for (int i = 0; i < num_per_tile; ++i){
7      if (> alpha_dest 0.01) {
8         dataSpl elem = lists[tile_index][i];
9         ...
10        }
11     }
12     else
13         break
14 }
```

Figure 5: Feature evaluation: Depending on the configuration or targets used, highlighted in green (top), the same features, highlighted in orange, expand into different resulting code (bottom)

and are mapped straightforwardly to one single element previously described in the implementation space in Section 4.

More sophisticated features are shown in Figure 8. These features are used together to assemble one single variation inside the implementation space. The given example generates code that iterates chunk-wise over an input list (*loop-over-blocks*, lines 1 and 11)), after which the inner loop processes the single chunk elements (*process-blocks*, lines 6, 22, 26). *Process-blocks* is not used directly within the feature-implementation of *loop-over-blocks*, but appears later on, inside its body as seen in line 3 in Figure 7. Processing a list chunk-wise requires a special case handling, since the last iteration only processes the residual list elements.

One possibility to manage this is to check each iteration step whether the currently processed chunk is the last one and to set the upper limit of the inner loop according to the number of remaining elements. A respective implementation is shown in the upper part (line 1 to 9) of Figure 8. In this case, *loop-over-blocks* does not distinguish between the first blocks ($0 <= i < iterations$) and the last one ($i == iterations$), thus iterates over all chunks. The associated feature (*process-blocks*, line 6), then limits the upper bound for the inner loop (N) by setting it either to the standard block-width (*elems-per-iter*) or, if the outer loop reached the last element, to the size of the last chunk (*last-iter-width*). In short, *loop-over-blocks* iterates over the complete range of chunks and *process-blocks* tests whether the last chunk is to be processed to set suitable limits for the inner loop.

A different approach handling the last chunk is shown in the lower part (line 11 to 27) of Figure 8. The underlying concept is to process the last chunk separately. Consequently, the outer loop iterates over every chunk except for the last one, which must be handled individually outside the loop. The benefit of this method is that the conditional assignment (line 7) can be omitted and the necessary *proces-blocks* features (line 21 to 27) reduce their complexity. Yet, as it can be seen, we now employ two, slightly different *proces-blocks* features, which depend on different targets. One deploys a loop, which iterates over the full range of the block size, and the other one processes the width of the final chunk.

```
1  (defmacro instantiate (config)
2    `(with-config ,config
3       (for ((int i 0) (< i num-per-tile) ++i)
4         (early-out (> alpha-dest 0.01)
5           ...
6
7  (instantiate (make-config no-early-out <do-this> ...))
8  (instantiate (make-config blockwise-early-out <skip-that> ...))
```

Figure 6: Single implementation for multiple variants

```
1  (with-iteration-bounds
2    (loop-over-blocks ,config
3      (early-out (> alpha-dest 0.01f)
4        (process-blocks
5          (load-current-element elem
6            (decl (((dist-type) dist-x (dist x))
7                   ((dist-type) dist-y (dist y))
8                   ((coc-type-in-node) coc-sq
9                                       (funcall elem.sq-coc)))
10             (if (<= (+ (* dist-x dist-x)
11                        (* dist-y dist-y)) coc-sq)
12               (with-alpha
13                 (set-color x r)
14                 (set-color y g)
15                 (set-color z b)
16                 (set-sample)))))))
17     (sync-after-iteration)))
```

Figure 7: Multidimensional implementation with features

To fuse the looped code with that of the last chunk, having been processed separately, both of the described *process-blocks* features need to be used. Since the algorithmic procedure for every block is the same, the body of the *loop-over-blocks* feature can be duplicated and used for both parts sequentially. The corresponding implementation (lines 11 to 19) simply duplicates the forwarded body, whereby one is placed within the for-loop (line 16) and the other is appended afterwards.

Both bodies are implemented at the same and cannot be changed usefully by means of list modification at this point. However, they still must generate different code. Therefore, we substitute the currently valid configuration individually for each body. Since it is not desirable to overwrite the configuration completely, the newly introduced configurations are composed of the targets previously passed in (*config*, line 15 and 18). These are already being used for the global configuration, and extended by the respective, locally required targets (*full-block*, line 15 and *peel-block* line 18). Eventually we have two nearly equal sections, which expand into two different specific implementations.

These examples were aimed at providing further insight into how we approached the design of a merged implementation for all meaningful variants previously described in Section 4.

In the following section we will evaluate each instance in terms of their performance in order to identify the optimal combination.

```
(implement loop-over-blocks (check-residual-block)
  `(for ((int i 0) (<= i iterations) ++i)
          (load-threads-local-data i)
    ,@body))

(implement process-blocks (check-residual-block)
  `(decl ((int N (? (== i iterations) last-iter-width
       elems-per-iter)))
    (loop-over-loaded-block (j N)
    ,@body)))

(implement loop-over-blocks (peel-residual-block)
  `(progn
    (for ((int i 0) (< i iterations) ++i)
      (load-threads-local-data i)
      (with-config (make-config ,@config full-block)
        (progn ,@body)))
    (load-threads-local-data iterations)
      (with-config (make-config ,@config peel-block)
        (progn ,@body))))

(implement process-blocks (full-block)
  `(loop-over-loaded-block (j elems-per-iter)
    ,@body))

(implement process-blocks (peel-block)
  `(loop-over-loaded-block (j last-iter-width)
    ,@body))
```

Figure 8: Implementation for checking (top) and peeling (bottom) residual list elements.

6. EVALUATION AND RESULTS

In this section we briefly evaluate the generator and resulting code of our generic implementation on a Nvidia Geforce GTX Titan, 980, and 980Ti graphics card. We strive to analyze each possible combination of aspects and GPU architecture to identify substantial coherences and special cases. However, to cover every generated accumulation loop, we have to consider 320 versions and testing all of them on three GPUs results in 960 individual measurements.

All time measurements in the context of implementation aspects and architectures are shown in Figure 9. It should be noted that the visualization is normalized, thus the lower bound is prepresenting the shortest processing time and the upper bound the longest. In addition, for each graphics card different values for upper and lower bounds were applied. This representation was chosen due to better exposition of specific patterns.

Performance Evaluation. The most performant combination for the Maxwell architecture (980 and 980Ti) is highlighted in green; for the Kepler architecture (Titan), it is highlighted in blue. Interestingly, the latter is as well suitable for the Maxwell, but not vice versa. Despite the fact that the best Titan measurement is only 0.1 ms faster than the slowest GTX 980Ti measurement, all GPUs share favorable implementation aspects. The resulting code of the best implementation for the Maxwell architecture can be found in the appendix in Figure 10.

A surprising insight is that it is not always the best choice, as initially assumed, to implement an early-out behaviour. As can be seen, all version highlighted in violet, the most aggressive early-out method, are in most cases slower than versions highlighted in yellow, which are implemented with only one exit attempt per processed block, and versions highlighted in red, which are implemented completely without early-out mechanism. This being said, applying a Maxwell GPU, a moderate early-out technique seems to be a better choice over omitting early-out completely. Employing a Kepler GPU, early-out techniques should be avoided, at least in our use case.

Further discoveries are the superior performance for the Maxwell architecture, when the last work unit is processed separately, and for both architectures, when using float instead of integer values to compute the distance, even though this requires additional conversions with a *__half2float()* call. The remaining aspects seem to have only a minor or no impact at all.

In contrast to the plot of the GTX Titan, it is striking how similar the measurement charts of the GTX 980 and the GTX 980Ti appear even though they represent different performance ranges. Although noticeable, it is not very surprising, since both GTX 980 and 980Ti share the same architecture.

Code Evaluation. Our preliminary objective was a general implementation of the accumulation loop of our depth of field algorithm [19]. At first we extended the initial concept with simple features, followed by a few iterations of including and testing newly emerging variation possibilities leading to an implementation that provides 320 versions of our algorithm.

Extending, debugging, and maintaining has been done with ease, since, per variant only one location of the program code has to be considered and modifications affect generated instances of features globally. This behaviour is an additional benefit in itself, precisely because we now can guarantee that each specific instance of the generated feature aspect is identical, in contrast to manually copied and modified code. With this assurance we do not risk to draw the wrong conclusion comparing unequal or faulty code instances and are able to keep many variations, which, as the comparison across the architecture generations shows, can be beneficial in the long run.

The final C-Mera code consists of 300 lines (275 without comments) of feature implementations and expands into over 19000 lines (16000 without comments) of CUDA code that provides 320 distinct kernels.

7. CONCLUSION

In this paper, we showed how we were able to discard redundant instances of essentially similar code fragments by merging them into general structures. Instead of implementing the whole algorithm for each of its possible variants, the algorithm is implemented only once and every ambiguity is replaced with its respective feature.

This approach enabled us to simply unfold, examine, and maintain 320 different versions of the same algorithm to eventually determine the best fitting feature-set in terms of performance for specific GPUs and architectures.

By fully expanding every possible combination we were able to analyze and maintain a much broader range of measurements and identify unexpected findings. In addition, we can securely rely on the accuracy of each kernel, since the expansion of individual feature is globally consistent and investigation on the output code can be done with ease.

In conclusion it can be said that employing a feature oriented programming methodology can proof quite useful when it comes to exploring and analyzing a vast amount of possibly suitable variants, especially when interesting future variations are to be expected.

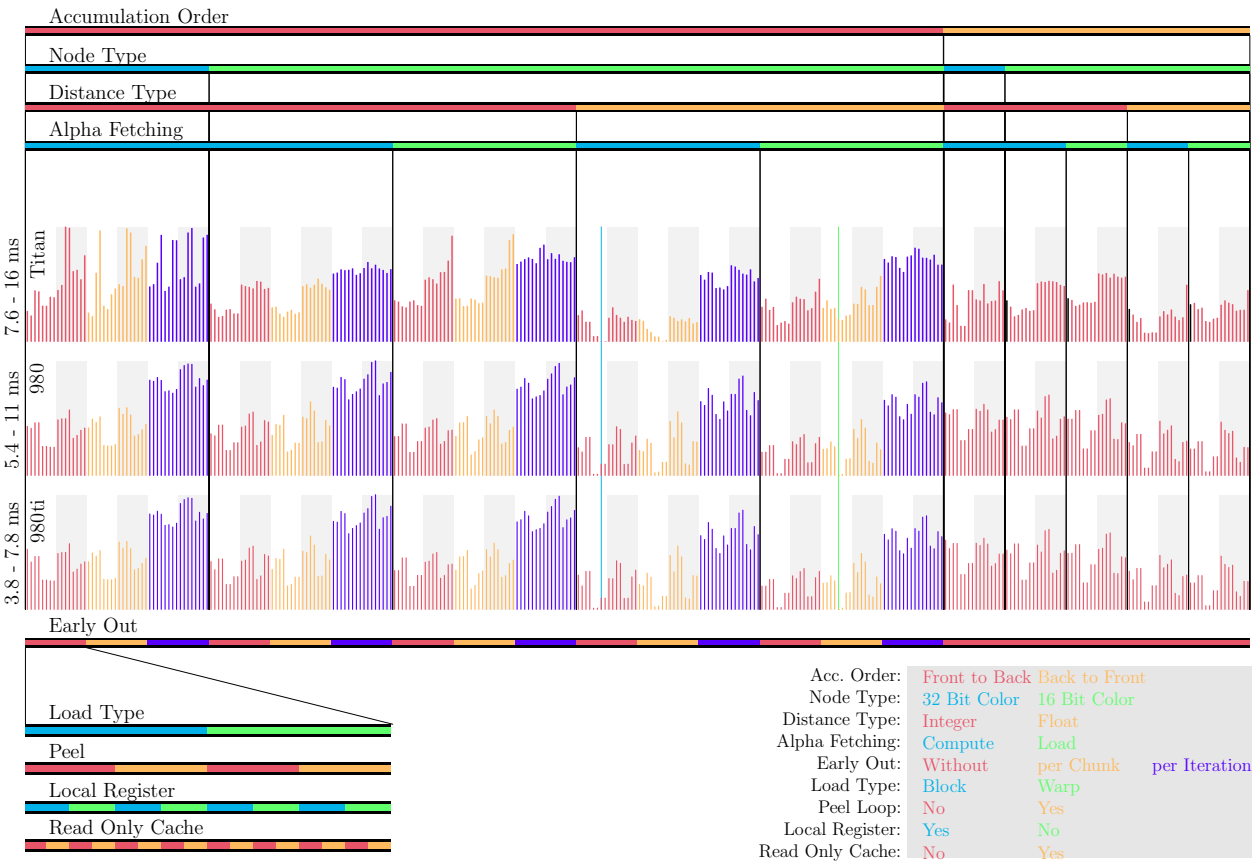

Figure 9: Individual measurements of various feature combinations on 3 GPUs.

Acknowledgments

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

8. REFERENCES

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development, July/August 2009. Refereed Column.

[2] S. Apel and C. Kästner. Virtual Separation of Concerns - A Second Chance for Preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[3] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.

[5] J. Demers. Depth of field: A survey of techniques. In R. Fernando, editor, *GPU Gems*. Pearson Higher Education, 2004.

[6] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM.

[7] A. Fredriksson. Amplifying C. http://voodoo-slide.blogspot.de/2010/01/amplifying-c.html, 2010.

[8] K. O. W. Group. *The OpenCL Specification*, March 2016.

[9] T. Harada, J. McKee, and J. C. Yang. Forward+: Bringing deferred lighting to the next level. In *Eurographics 2012 - Short Papers Proceedings, Cagliari, Italy, May 13-18, 2012*, pages 5–8, 2012.

[10] S. E. Keene. *Object-oriented programming in COMMON LISP - a programmer's guide to CLOS*. Addison-Wesley, 1989.

[11] J. Kessenich, D. Baldwin, and R. Randi. *The OpenGL Shading Language*, January 2014.

[12] J. Krivanek, J. Zara, and K. Bouatouch. Fast depth of field rendering with surface splatting. In *Computer Graphics International, 2003. Proceedings*, pages 196–201. IEEE, 2003.

[13] M. McCool, S. Du, T. Tiberiu, P. Bryan, and C. K. Moule. Shader algebra. *ACM Transactions on Graphics*, pages 787–795, 2004.

[14] R. Membarth, P. Slusallek, M. Köster, R. Leißa, and S. Hack. High-performance domain-specific languages for gpu computing. GPU Technology Conference

(GTC), March 2014.

[15] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, September 2015.

[16] M. Potmesil and I. Chakravarty. A lens and aperture camera model for synthetic image generation. In *Proceedings SIGGRAPH 1981*, pages 297–305. ACM, 1981.

[17] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Lecture Notes in Computer Science, pages 419–443. Springer-Verlag, June 1997.

[18] G. Riguer, N. Tatarchuk, and J. R. Isidoro. Real-time depth of field simulation. In W. Engel, editor, *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*. Wordware, Plano, Texas, 2003.

[19] K. Selgrad, L. Franke, and M. Stamminger. Tiled Depth of Field Splatting. In J. Jorge and M. Lin, editors, *Eurographics 2016 – Posters*. The Eurographics Association, 2016.

[20] K. Selgrad, A. Lier, J. Dörntlein, O. Reiche, and M. Stamminger. A High-Performance Image Processing DSL for Heterogeneous Architectures. In *Proceedings of ELS 2016 9rd European Lisp Symposium*, pages to–appear, New York, NY, USA, 2016. ACM.

[21] K. Selgrad, A. Lier, F. Köferl, M. Stamminger, and D. Lohmann. Lightweight, generative variant exploration for high-performance graphics applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 141–150, New York, NY, USA, 2015. ACM.

[22] K. Selgrad, A. Lier, M. Wittmann, D. Lohmann, and M. Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *Proceedings of ELS 2014 7rd European Lisp Symposium*, pages 80–87, 2014.

[23] P. Slusallek and I. Georgiev. Rtfact: Generic concepts for flexible and high performance ray tracing. In R. J. Trew, editor, *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008*, pages 115–122, RT08 Reception Warehouse Grill 4499 Admiralty Way Marina del Rey, CA 90292, 2008. IEEE Computer Society, Eurographics Association, IEEE.

[24] W. Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[25] G. Thomas. Compute-Base GPU Particle Systems, 2014. GDC'14.

[26] T. Veldhuizen. Template metaprograms. *C++ Report*, May 1995.

APPENDIX

```
1   // This is DOF accumulation with the following features:
2   // - block-load-with-syncthreads
3   // - peel-odd-blocks
4   // - node-rgb16-c32-xy16
5   // - copy-current-entry-to-register
6   // - synced-blockwise-early-out
7   // - directly-load-from-global-memory
8   // - load-alpha
9   // - front-to-back
10  // - float-dist
11
12  __global__ void kernel_213(int width, int height, int2 tileWH, int2 tilesizes, dataSpl
        **lists, float max_coc, uint *atomic_counters, ushort4 **hdr_output)
13  {
14          int2 gid = make_int2((blockIdx.x * blockDim.x) + threadIdx.x, (blockIdx.y *
                blockDim.y) + threadIdx.y);
15          if ((gid.x >= width) || (gid.y >= height))
16                  return;
17          float2 gidF = make_float2(float(gid.x), float(gid.y));
18          int tile_index = (gid.x / tileWH.x) + (tilesizes.x * (gid.y / tileWH.y));
19          int num_per_tile = (tileWH.x * tileWH.y) + ((int)atomic_counters[tile_index]);
20          float4 color = make_float4(0, 0, 0, 1);
21          int tid = (threadIdx.y * 16) + threadIdx.x;
22          __shared__ dataSpl local_data[256];
23          float alpha_dest = 1.0f;
24          const int elems_per_iter = 256;
25          int iterations = num_per_tile / 256;
26          int last_iter_width = num_per_tile % 256;
27          for(int i = 0; i < iterations; ++i){
28                  local_data[tid] = lists[tile_index][(i * 256) + tid];
29                  __syncthreads();
30                  if (__any(alpha_dest > 0.01f)) {
31                          for(int j = 0; j < elems_per_iter; j += 1){
32                                  dataSpl elem = local_data[j];
33                                  float dist_x = __half2float(elem.x()) - gidF.x;
34                                  float dist_y = __half2float(elem.y()) - gidF.y;
35                                  float coc_sq = elem.sq_coc();
36                                  if (((dist_x * dist_x) + (dist_y * dist_y)) <= coc_sq) {
37                                          float alpha = __half2float(elem.w());
38                                          color.x += (alpha_dest * alpha *
                                                    __half2float(elem.r()));
39                                          color.y += (alpha_dest * alpha *
                                                    __half2float(elem.g()));
40                                          color.z += (alpha_dest * alpha *
                                                    __half2float(elem.b()));
41                                          alpha_dest = (1.0f - alpha) * alpha_dest;
42                                  }
43                          }
44                  }
45                  else
46                          break;
47                  __syncthreads();
48          }
49          local_data[tid] = lists[tile_index][(iterations * 256) + tid];
50          __syncthreads();
51          if (alpha_dest > 0.01f)
52                  for(int j = 0; j < last_iter_width; j += 1){
53                          dataSpl elem = local_data[j];
54                          float dist_x = __half2float(elem.x()) - gidF.x;
55                          float dist_y = __half2float(elem.y()) - gidF.y;
56                          float coc_sq = elem.sq_coc();
57                          if (((dist_x * dist_x) + (dist_y * dist_y)) <= coc_sq) {
58                                  float alpha = __half2float(elem.w());
59                                  color.x += (alpha_dest * alpha *
                                            __half2float(elem.r()));
60                                  color.y += (alpha_dest * alpha *
                                            __half2float(elem.g()));
61                                  color.z += (alpha_dest * alpha *
                                            __half2float(elem.b()));
62                                  alpha_dest = (1.0f - alpha) * alpha_dest;
63                          }
64                  }
65          __syncthreads();
66          hdr_output[gid.x][gid.y] = make_half4(color.x / (1.0f - alpha_dest), color.y /
                (1.0f - alpha_dest), color.z / (1.0f - alpha_dest), 1.0f);
67  }
```

Figure 10: C-Mera generated kernel with the best performance on Nvidia GTX 980Ti.