Explicit Dependence Metadata in an Active Visual Effects Library

Jay L. T. Cornwall[1], Paul H. J. Kelly[1], Phil Parsonage[2], and Bruno Nicoletti[2]

[1] Imperial College London, UK
[2] The Foundry, UK

Abstract. Developers need to be able to write code using high-level, reusable black-box components. Also essential is confidence that code can be mapped to an efficient implementation on the available hardware, with robust high performance. In this paper we present a prototype component library being developed to deliver this for industrial visual effects applications. Components are based on abstract algorithmic skeletons that provide metadata characterizing data accesses and dependence constraints. Metadata is combined at run-time to build a polytope representation which supports aggressive inter-component loop fusion. We present results for a wavelet-transform-based degraining filter running on multicore PC hardware, demonstrating 3.4x–5.3x speed-ups, improved parallel efficiency and a 30% reduction in memory consumption without compromising the program structure.

1 Introduction

Component-based programming is a software development paradigm in which interoperable and composable components are written, tested and debugged in isolation of one another. They can then be composed into useful programs, perhaps from a library of reusable components. This idea comes so naturally that it has become the primary mode of user interaction in professional video compositing applications, where the user composes effects and video clips into workflows. Elegant design comes at a price, however, and the goals of component-based programming are frequently at odds with performance.

In this paper we explore the barriers to high performance in an industrial visual effect by building a dynamic, self-optimising library from its constituent algorithms. At the heart of our library is the concept of dependence metadata, which enables complex code transformations without expensive dependence analyses. We focus on an effect called degraining [21], produced by our industrial collaborators The Foundry, designed to suppress the random texturing noise introduced by photographic film without compromising an image's clarity. This is achieved by first analysing the grain, or by matching it against a database of grain patterns, and then applying a wavelet-based removal algorithm. The latter is more computationally intensive and thus forms the focus of our work.

This work was partly funded by the EPSRC (ref EP/E002412).

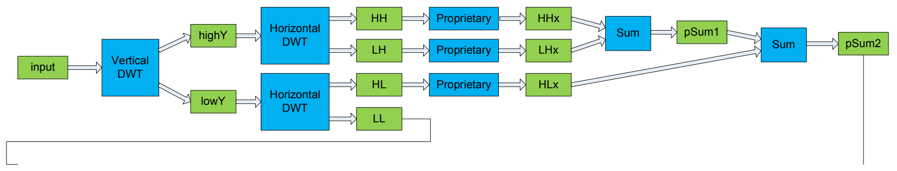

Fig. 1. One iteration of degraining in component form, replicated four times with an appropriate terminator. Dark boxes represent components, while light boxes represent data handles. Handles feed component outputs to other inputs without an intermediate data set necessarily existing, unless the programmer explicitly evaluates the handle.

Figure 1 shows a breakdown of one iteration of the degraining algorithm into components. The complete algorithm chains this graph four times in succession. Our breakdown is faithful to the industrial codebase except that we split summation from the proprietary component, a small improvement in design that our optimising framework allows us to afford. Note that this will artificially inflate our performance gains somewhat, but we believe this to be the most desirable construction; emphasised by the difficulties we encountered in debugging the partially fused implementation. The original algorithm was written in a similar component-based structure; partly to promote reusability and to simplify debugging, and partly due to the difficulty of managing heavily fused code.

From a performance perspective, the optimum structure looks very different. Some knowledge of the dependence structures for each component reveals great redundancy in iteration over intermediate results. Each component is implicitly a whole new iteration. Large data sets carry information from one component to the next, spilling into higher levels of the memory hierarchy, when restructuring transformations could greatly reduce their size. Opportunities for instruction-level parallelism (ILP), an important tool in superscalar architectures, are limited by the barriers between the computations of each component. Crucially, none of these optimisations could be applied directly to the code without greatly disrupting the component-based design. This tension between good design and high performance tends to lead programmers to choose one at the expense of the other.

We argue that these optimisations are crucial for performance and that, with some innovative programming, they can be consistent with good design. Our solution avoids disruption in the original code by promoting a generative approach, in which components are equipped with functionality to create their own implementations. Problem-specific kernel code is left to the programmer as before, while we take control of the loops and collect high-level metadata describing each component's dependence structure. Delayed evaluation reveals component compositions and runtime code generation allows us to produce context-sensitive optimised implementations. The fundamental transformations leading to faster code are made safe and precise by dependence metadata. These are used to build a polytope representation of the loop nests from which optimised code can be instantiated.

In summary, the main contributions of this paper are:

– **Dependence metadata as a tool for optimisation**. In Section 2 we discuss the role and collection of dependence metadata in a component environment through algorithmic skeletons. This information enables precise loop shifting, loop fusion and array contraction without difficult analysis of the implementation.
– **Evaluation of an active visual effects library**. In Section 3 we present a complete active visual effects library built around a polytope code generation framework. We evaluate its performance with a component-based industrial visual effect.

2 System Design

An overview of our design is shown in Figure 2. We have chosen to adopt an offline phase in which optimised code is compiled and linked to the client application. This contrasts with other approaches that maintain client/library separation by moving this phase to runtime. Our approach benefits from requiring no build environment on the end-user system and from having no code generation or compile-time overhead. We lose the ability to specialise to dynamic parameters without pre-tracing every instance of them, but consider this to be a worthwhile trade-off given the large interactive variability of those parameters in our target applications. In practice, we only generate multiple traces when the component graph changes with a parameter (e.g. by disabling or reordering operations).

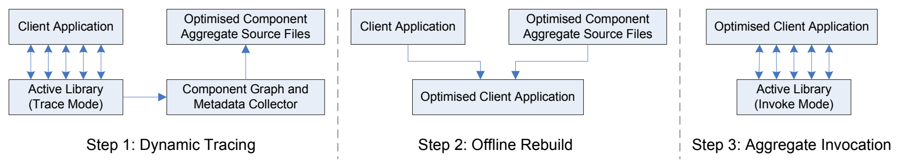
Client Application
Optimised Component Aggregate Source Files
Active Library (Trace Mode)
Component Graph and Metadata Collector
Step 1: Dynamic Tracing
Client Application
Optimised Component Aggregate Source Files
Optimised Client Application
Step 2: Offline Rebuild
Optimised Client Application
Active Library (Invoke Mode)
Step 3: Aggregate Invocation

Fig. 2. Stages of the optimisation workflow. The client is run with the library in a trace mode. Optimised aggregate library code is generated and embedded manually into the client application. A second library mode invokes this aggregate code in normal usage.

2.1 Library Front-End

We have designed the front-end to our active library with transparency in mind. The goal is to present an interface to the programmer which matches the existing imperative execution model, while retaining the flexibility to switch to a delayed evaluation mode. Our solution uses proxy functions to build a graph of components connected by abstract data handles at runtime, as shown in Listing 1.1. This code excerpt produces the graph for one iteration of degraining as shown in Figure 1. Run in an imperative mode, the same program would invoke the correct components in sequence and instantiate data handles with real data sets on-demand. This is useful for generating traces, invoking aggregate code and for debugging the optimisation engine without changing the client application.

```
/* Real data sets for reading/writing data. */
Handle input(new Image(width,height,components));
Handle pSum2(new Image(width,height,components));
Handle LL(new Image(width,height,components));

/* Virtual data sets for automatic instantiation. */
Handle highY,lowY,HH,LH,HL,HHx,LHx,HLx,pSum1;

VertDWT(input,highY,lowY,filterHeight,pass);
HorizDWT(highY,HH,LH,filterWidth,pass);
HorizDWT(lowY,HL,LL,filterWidth,pass);

Proprietary(HH,HHx);
Proprietary(LH,LHx);
Proprietary(HL,HLx);

Sum(HHx,LHx,pSum1);
Sum(pSum1,HLx,pSum2);
```

Listing 1.1. Component-based front-end with data sets, handles and proxy functions for a single iteration of degraining.

Components are constructed through an algorithmic skeleton interface. The goal of skeletons in our library is twofold. Firstly, we need to separate loops from programmer-written kernels so that transformations can be applied to the iteration space. Secondly, we need to extract high-level metadata describing an algorithm's dependence structure in order to determine which transformations can be applied and in what order. We make loose use of the skeleton terminology from Nicolescu and Jonker's work on skeletons in image processing [14] but do not distinguish the numbers of inputs or outputs; instead these parameterise the skeleton.

We place some constraints on the use of skeletons in our library for performance reasons. By controlling the loop structures we can enforce an iteration order and ensure that each element in the output is computed only once. The latter constraint may be relieved through a scatter skeleton but note that this will block loop fusion if the scatter distance is not limited to a subset of the output (i.e. it is not a global operation). The former constraint is somewhat configurable by the algorithm, in being able to choose forwards/backwards and horizontal/vertical iteration parameters. This provides enough flexibility in managing loop-carried dependencies to account for all of the components we have investigated so far, but we plan to explore more complex skeletons in the future.

Figure 3 classifies the three non-proprietary components of degraining as skeletons. Summation matches a simple point skeleton parameterised by two inputs and one output. There are two per-iteration data dependencies from the inputs to a corresponding point in the output. Both the vertical and horizontal DWT match the filter skeleton, parameterised by one input, two outputs and the direction and dimensions of the filter. Dependencies from the input to both outputs cover the filter area. As a result, the dependence structures of the horizontal and vertical DWT components change dynamically with the filter size parameter. Our optimisation engine accounts for this by generating aggregate loops with an iteration space parameterised by this variable.

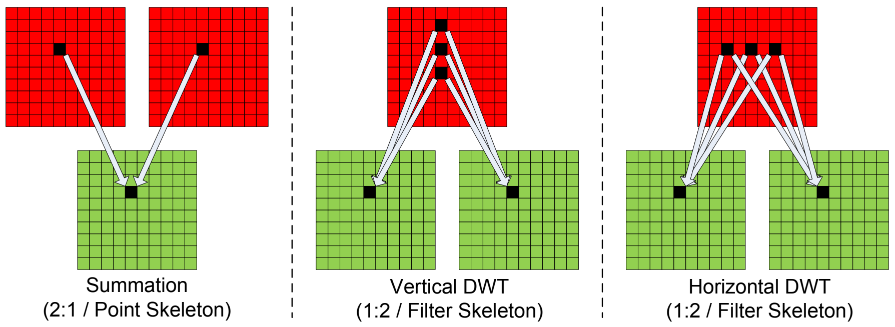

Fig. 3. Skeleton classifications for the non-proprietary components of degraining.

To illustrate the use of skeletons in our library, Listing 1.2 shows a partial implementation of the vertical DWT component. The component object subclasses an appropriate skeleton. The programmer provides a scalar kernel and an optional vector kernel, expressed in terms of the arrays *in1*, *out1*, and *out2*, and indices y, x and c. Our code generator is free to use whichever implementation it prefers but the current implementation will always choose the vector kernel if it can be used throughout the entire graph; otherwise scalar is chosen. *_filterHeight* is a parameter to the skeleton that is used inside the kernel and inside the *getRadius* function. The *getRadius* function encodes dependence metadata for the skeleton by defining a windowed access region over the *in1* array, centred over the current iteration point (x, y) – this is discussed in more detail in the following subsection. Thus the metadata is provided by the programmer through a simple overloaded function call in the skeleton.

2.2 Deriving Transformation Parameters from Metadata

We focus on loop fusion and array contraction [2] as potentially beneficial cross-component optimisations to apply to the component graph, as demonstrated in earlier work [18]. In order to apply these transformations safely we must derive two parameters: the loop shift required for fusion and the contracted size of intermediate data sets. Computing these parameters normally requires detailed region and liveness analyses. We aim to demonstrate that explicit dependence metadata can achieve the same result at a much lower cost.

```
class VertDWTSkel : public Filter1DSkeleton {
  void scalarKernel(...) {
    float valT = in1[y-(_filterHeight/2)][x][c];
    float valM = in1[y][x][c];
    float valB = in1[y+(_filterHeight/2)][x][c];
    out1[y][x][c] = (valM-(valL+valR)*0.5f)*0.5f;
    out2[y][x][c] = valM-out1[y][x][c];
  }

  void vectorKernel(...) {
    __m128 valT = in1[y-(_filterHeight/2)][x];
    __m128 valM = in1[y][x];
    __m128 valB = in1[y+(_filterHeight/2)][x];
    out1[y][x] = (valM-(valL+valR)*0.5f)*0.5f;
    out2[y][x] = valM-out1[y][x];
  }

  void getRadius(int *radius) {
    radius[0] = _filterHeight;
    radius[1] = 0;
  }
};
```

Listing 1.2. Partial implementation of the Vertical DWT.

Our metadata is inspired by the THEMIS proposal [12]. THEMIS mapped out a set of properties to describe a procedure's dependence structure. At each point in its iteration domain and for each operand to a procedure, a set of indices which may be read by the procedure is defined. Similarly for the data items which may be written to, further sets are defined. The precise representation of this information is left to the programmer. The authors give an example where affine functions are sufficient to represent dependencies for each iteration relative to the position in the iteration domain. We find that a similar approximation is suitable for all of the components in the degraining algorithm.

We assume that each skeleton's kernel writes once to all points in the output data set(s). This is not the case for our "scatter" skeleton (not used in this algorithm), which potentially overwrites a single point many times, but in that case we simply introduce a larger shift to ensure that we do not read values from the preceding component until they have permanently left the scatter window. Dependence metadata is defined as the dimensions of a window centred over corresponding points in all of the input data sets. For a point skeleton this is simply (1,1). The vertical filter skeleton will have dependence metadata represented by (1,n), where n is the filter height, while the horizontal filter skeleton is similarly characterised by (n,1). This simple scheme is sufficient to enable computation of the transformation parameters for maximal fusion across the entire degraining algorithm.

For a detailed explanation of deriving optimal array contraction parameters see [20]. However, our approach allows these parameters to be derived trivially. Their value is equal to the loop shift of the succeeding component (i.e. precisely the number of iterations that the intermediate data should be held for). We use a small optimisation trick in computing the contracted size by noting that indexing a contracted array requires an expensive modulus operation. However, by padding the contracted size to the nearest power-of-two, cheap bitwise operations can be substituted for modulus arithmetic.

Finally, we put all of this together with a simple propagative algorithm that walks the component graph, computing loop shifts and contracted data set sizes from the *getRadius* dependence metadata and the input transformation parameters to each component.

2.3 Code Generation

Our code generation scheme is slightly unusual. While it would be trivial to generate some text representing the shifted loops, fusion is a difficult transformation to apply. By recognising loop fusion as an iteration space scanning problem – that is, to consolidate kernels in common slices of a domain – we leverage the polytope model for a solution. Polytopes are a mathematical formulation of a loop nest, its statements and their dependence. Loop fusion in this model is trivially solved by overlapping multiple polytopes.

We make use of the CLooG (Chunky Loop Generator) library [3] to achieve this. CLooG is a loop generation tool based on the polytope model. It devises an iteration scheme to visit all of the integral points in a polyhedron under a system of scheduling constraints. Of the many possible loop nests that arise, CLooG picks the one most optimised in control flow. A side effect of this choice is that loop unrolling and fusion are applied implicitly in the polyhedral scanning process. CLooG will not perform enabling transformations, such as loop shifting, by itself. Instead, we provide the library with pre-shifted iteration spaces and kernels with shifted and contracted array indexing, along with a guarantee that no loop-carried dependencies exist between statements of different kernels.

CLooG requires a client code generator to fill the loops it generates with appropriate kernels. The client supplies a unique identifier for a kernel when creating a polytope, and is provided with sequences of the same identifiers during fused code generation. One way to capture programmer-written kernels from the target application, as demonstrated in the TaskGraph [4] library, is to use template metaprogramming to build a high-level representation of the kernel which can later be unparsed back to text. This approach provides a semantic advantage and opportunities to modify the kernel. However, it imposes a syntactic structure that is limited in flexibility and familiarity to the programmer. We chose a less intrusive approach, using a simple pre-processing script to copy kernels from C++ source files into strings within the skeleton classes. In the future we could use source-to-source translators, such as ROSE [19], to perform optimisations on the string-based kernels.

Kernel chaining and array contraction are applied in a pattern matching pre-generation pass. The *inN* and *outN* references from programmer-defined kernels are chained together with unique arrays called *_named_arrX*. These arrays are instantiated with the contracted sizes computed in Section 2.2. They are freed after the loops have finished. User-supplied input and output arrays are referenced directly and are not involved in contraction. Listing 1.3 shows a fragment of CLooG's output for the degraining algorithm. Loop shifting, unrolling (not shown here), fusion and array contraction have all taken place to orchestrate the fully optimised algorithm. In the most aggressively fused case, the complete listing extends to over fifteen thousand lines of code, ninety loops and numerous unrolled fragments.

```
for (y=29;y<=paddedHeight-16;y++) {
  ...
  for(x=6;x<=9;x++) {
    // Vertical DWT
    {__m128 vValT = _mm_load_ps(&_named_arr0[y-1][x][0]);
    __m128 vValM = _mm_load_ps(&_named_arr0[y][x][0]);
    __m128 vValB = _mm_load_ps(&_named_arr0[y+1][x][0]);
    _named_arr2[i&3] = (vValM-(vValT+vValB)*vPoint5)*vPoint5;
    _named_arr3[i&3] = vValM-_named_arr2[i&3];}
    ...
    // Vertical DWT
    {__m128 vValT = _named_arr5[((-2-2)*paddedWidth+i)&32767];
    __m128 vValM = _named_arr5[((-2)*paddedWidth+i)&32767];
    __m128 vValB = _named_arr5[((+2-2)*paddedWidth+i)&32767];
    _named_arr6[i&7] = (vValM-(vValT+vValB)*vPoint5)*vPoint5;
    _named_arr7[i&7] = vValM-_named_arr6[i&7];}
  }
}
```

Listing 1.3. A fragment of CLooG's output for degraining.

3 Experimental Results

The degraining algorithm is implemented in C++ with our skeleton optimisation framework. Two implementations are considered throughout this chapter: one written with scalar operations and the other with SSE intrinsics. This computation is trivially parallelised by statically partitioning the image to utilise all cores of a multicore system. We allow the compiler to vectorise the scalar code as it sees fit, but in practice it is able to do very little. Our target compiler is Intel C/C++ 10.0.025 on the Linux 2.6 operating system, in 64-bit mode where processor support was available. A brief comparison with GCC 4.1.2 showed this to be the favourable choice for performance on all benchmarking systems. We use the flag set '-O3 -funroll-loops' and append an architecture-specific optimisation flag as recommended by the manual – using -xW for non-Intel processors.

Before looking at the experimental results it is worth noting a design decision which impacts performance throughout this chapter. All of our benchmarks operate upon three-component interleaved RGB single-precision floating-point data. In order to simplify the vector processing front-end, we chose to pad this data to RGBA with an unused alpha channel in the SSE intrinsic implementations. This raises memory pressure over the scalar implementations and introduces significant redundant computation. One alternative design that we considered involved separating colour channels into contiguous regions. Another used loop unrolling to process RGBR, GBRG and BRGB pixel fragments. Both of these approaches relieve memory pressure but complicate the front-end or back-end of our optimisation framework. We leave these considerations for future work.

3.1 Baseline Performance

Figure 4 introduces the baseline performance of our algorithm in scalar and SSE intrinsic forms. A spectrum of benchmarking platforms spreads the observed throughput to between 1 MPixel/s and 4 MPixels/s for useful image sizes. There is a clear reduction in performance on three out of four systems with the SSE implementation. In spite of the greater computational performance of SSE instruction units, memory performance dominates and suffers from the 33% larger RGBA pixels. On the Xeon, this almost perfectly correlates to a 33% drop in perfomance as the eight-core compute-heavy architecture is largely memory bound in this algorithm.

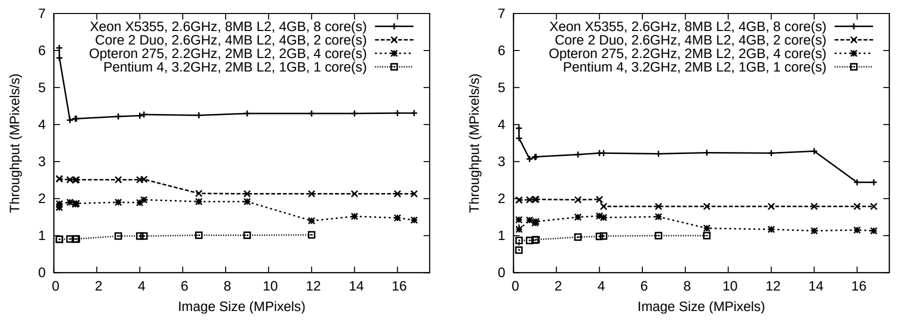

Fig. 4. Baseline throughput for scalar (**left**) and SSE intrinsic (**right**) implementations of degraining on interleaved RGB data for a range of practical image sizes. The SSE implementation pads RGB to RGBA before processing and unpads afterwards.

We build upon this data by recording the significant memory allocations and deallocations made by the algorithm during its lifetime. A single image size of 4000x3000x3 is used for comparison in later subsections; measurements scale proportionally to other image sizes. Peak memory consumption is a performance-limiting factor here – 970MiB for the scalar implementation and 1210MiB for SSE. This clearly demonstrates the padding that has occurred in order to simplify SSE application. Correlating this information with Figure 4 explains the absence of data for the Pentium 4 on images larger than 12 MPixels. The peak memory consumption exceeds the benchmarking system's capacity, resulting in page swapping and unstable performance. We omit data points where this has occurred due to the difficulty in obtaining representative samples.

3.2 Fusion within a Single Iteration

We now explore the benefits of loop fusion and array contraction within a single iteration of degraining. In fact, these transformations could also be applied across iterations of the algorithm to achieve maximal fusion. This comes at the expense of large loop shifts and an explosion in loop fragments in the output code, however. The impact of these factors is explored in Section 3.3, but we begin by constraining our transformations to a single iteration of the algorithm.

Figure 5 reports degraining performance with loop shifting and fusion applied to all components in the graph. First we show results of fusiona alone; shortly we show the impact of array contraction. Speed-ups are reported relative to the faster baseline results from Figure 4 – the scalar implementations in this case. Fusion is a risky optimisation in a component environment because it displaces the deallocations of temporary data from in-between loops. The fused loop nest accumulates a large number of allocations beforehand, leading to a 230% increase in peak memory consumption. Relative speed-ups are unimpressive and in fact degraded throughput has occurred in several cases.

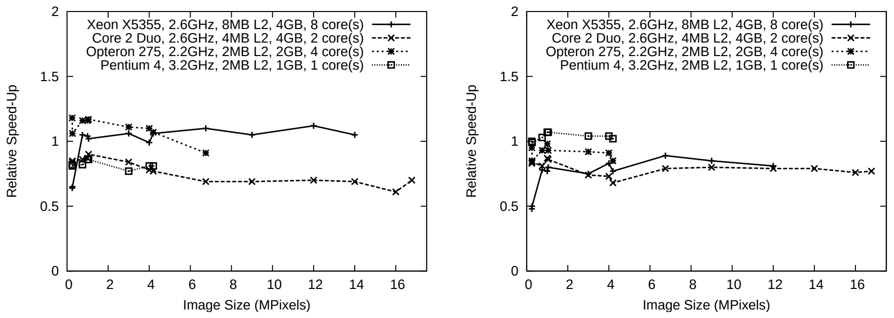

Fig. 5. Fused (but not contracted) speed-ups for scalar (**left**) and SSE intrinsic (**right**) implementations of degraining on interleaved RGB data, relative to the faster baseline implementations in Figure 4. The SSE implementation pads RGB to RGBA before processing and unpads afterwards.

Loop fusion is not applied in vain, however. By consolidating kernels inside a single loop nest, the transformation enables an array contraction optimisation. There is only a need to hold intermediate data for the duration of its reuse distance. We can communicate this information to the compiler by explicitly reducing the size of connecting data sets and by wrapping accesses to their arrays inside the contracted size. Figure 6 shows the final optimised speed-ups of degraining with loop shifting, fusion and array contraction applied. Performance is very positive in the scalar implementation with speed-ups ranging from 1.6x to 4.8x. Peak memory consumption has been reduced by 30% over the original implementation. Interestingly, the SSE implementation now begins to show promise with speed-ups between 3.4x and 5.3x.

3.3 Fusion across Multiple Iterations

In the preceding section we chose to arbitrarily constrain fusion to within one iteration of the degraining algorithm. We now explore the effects of fusion across

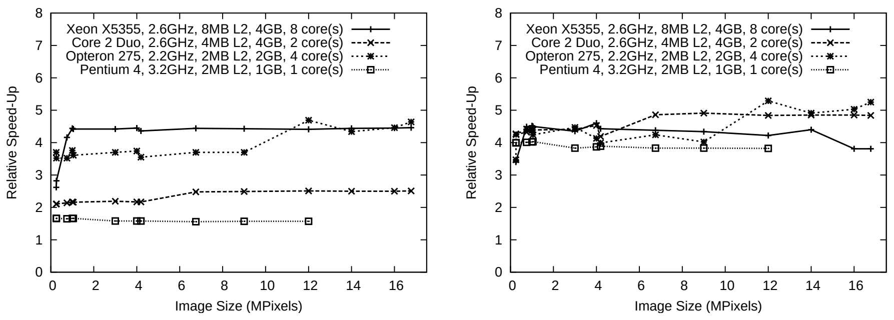

Fig. 6. Fused and contracted speed-ups for scalar (**left**) and SSE intrinsic (**right**) implementations of degraining on interleaved RGB data, relative to the fastest baseline implementations in Figure 4. The SSE implementation pads RGB to RGBA before processing and unpads afterwards.

multiple iterations, right up to complete fusion of the component graph. Crucially, fusing between iterations will result in loop shifts rising from 60KB to nearly 1MB because the vertical DWT component has a row-striding window. In addition, the amount of generated code grows superlinearly with the number of fusions applied. These effects result in larger working sets, greater register pressure and poorer instruction cache performance. Nevertheless, two large intermediate data sets can be contracted per iteration, following loop fusion, to improve memory performance.

Figure 7 presents results for fusion and contraction across one, two, three and four iterations of the algorithm. Only SSE intrinsic implementations are considered here, since they gave better average speed-ups in the preceding section. We find that performance isn't affected significantly in most cases. The Xeon system sees a small improvement with two fused iterations over one and experiences a similar drop from three to four fused iterations. We speculate that the heavily memory bound system benefits from inter-iterative contraction and suffers less from inflated working sets with its large L2 cache. The optimum average case fusion appears to be at two iterations.

3.4 Impact on Multicore Scalability

A final experimental analysis concerns the scalability of the pre- and post-optimised algorithm. The prevalence of multicore architectures places great emphasis upon scalability for current and future performance gains. Our optimisations do not target this factor directly, but may indirectly shift scaling bottlenecks by reducing memory pressure.

Figure 8 graphs the throughput of four implementations of degraining on the Xeon system as they scale up to eight cores. These have been fully fused within iterations but not between. An ideal result here would be linear scalability, but contention for shared resources and redundant processing at the edges – a side

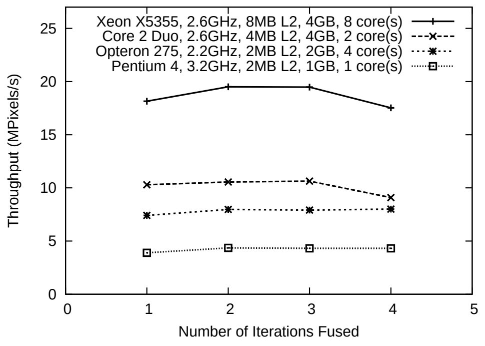

Fig. 7. Fusion and contraction within and across iterations of the SSE intrinsic implementation of degraining. At the fourth level of fusion, the entire algorithm is contained within a single aggressively fused loop nest.

effect of naive data parallelism, albeit small compared to the full data set – results in sublinear scalability in all cases. The post-optimised scalar implementation achieves closer to linear scalability than either pre-optimised case. However, the post-optimised SSE implementation experiences poor scalability after only two cores. It is worth noting that both implementations achieve roughly the same throughput with large numbers of cores – as both hit the memory wall – while SSE gives substantial improvements when fewer are in use.

Explaining these results is difficult because we have no direct method to determine which data points are CPU or memory bound. We believe that memory pressure is much lower in the optimised case, hence the large speed-ups, but that the algorithm remains memory bound. There is some indirect evidence to support this. Scalability is better when using scalar operations, particularly in the optimised case. Overall performance is of course lower but the algorithm scales more smoothly on a per-core basis. This is because SSE trades memory bandwidth for higher computational performance, so the vectorised cases exhibit high per-core performance but hit memory bottlenecks much sooner. Additional evidence comes from the speed-ups gained from vectorisation: 2.1x with one core in use and a little under 1.0x with eight cores. Padded data in the SSE implementation allows this figure to drop below one.

4 Related Work

Cross-component optimisation encompasses a spectrum of interprocedural techniques including data placement [6], loop transformation [1, 18] and implementation selection [11]. The key challenge is to tunnel across the execution and code visibility barriers present in a component-based programming model without compromising the program structure. Two enabling technologies, delayed evaluation [5] and runtime code generation [5, 4], have been demonstrated as effective and attractive infrastructure for cross-component optimisation [15]. Generative

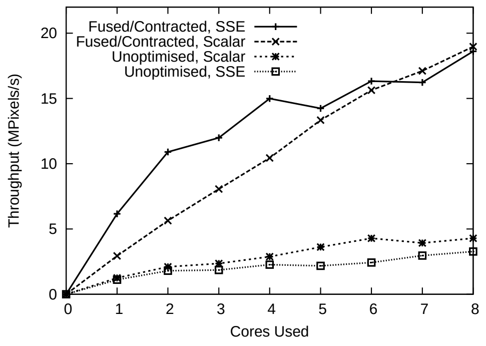

Fig. 8. Scalability of four implementations of degraining for a 4000x3000x3 image on a Dual Xeon X5355 2.6GHz (8 cores) with 8MB L2 cache and 4GB RAM in total.

programming is a paradigm which encapsulates this functionality into metaprogrammed self-optimising libraries [22, 10], termed active libraries. Kelly et al. proposed a metadata scheme [12] to carry information about component dependence to an optimising engine. This information is critical in ensuring correctness in code restructuring and efficiency in parallel data placement optimisations.

Algorithmic skeletons separate the problem-specific details of an algorithm, expressed in the full power of the underlying language, from structural features such as data dependence and iteration order. Skeletons have been researched extensively in parallel computing – as surveyed in [17] – as a programming model with explicit parallelism and communication semantics. Benoit et al. later refined the model to incorporate context-sensitive selection of operational parameters [7]. Adobe's Generic Image Library [8] is an implementation of the skeleton concept in the domain of image processing, enhancing fundamental data types with colour information and providing relevant algorithmic patterns.

Polytopes, in the context of software optimisation, are a mathematical formulation of loops, statements and dependence. In his seminal work on loop parallelisation [13] Lengauer illustrated the decomposition of a program into the polytope model and scheduling transformations to satisfy different processing goals. Code generation is a polyhedral scanning problem surveyed by Bastoul in [3] and incorporated into the CLooG library. Ongoing work by Pop et al. focuses on the integration of polytope transformations, through the CLooG library, into the GCC compiler [16]. Cohen et al. achieved similar integration with the Open64/ORC compiler [9], citing benefits in finding transformation sequences.

5 Conclusions and Further Work

In this paper we presented a visual effects library which takes an active role in the cross-component loop and data optimisations in a client application. We demonstrated the role of dependence metadata in replacing the complex program analyses previously required to apply these code transformations safely.

Algorithmic skeletons underpin our metadata collection interface and proved flexible enough to annotate all of the components in the degraining algorithm. We implemented a code generation framework in the polytope model with the CLooG library, which proved robust enough to correctly generate over fifteen thousand lines of code and ninety loops in the most aggressively fused case.

Our evaluation showed that loop shifting and loop fusion alone were not sufficient to make gains in performance, and in many cases resulted in degraded throughput due to inflation of the memory profile. Array contraction substantially improved memory performance thereafter, giving 3.4x–5.3x speed-ups in the SSE vector implementation. Peak memory consumption was reduced by 30% as a side effect of this transformation. We explored the impact of our optimisations on multicore scalability and demonstrated closer to linear scalability in the post-optimised case. The SSE vector implementation initially scaled better but hit the memory wall after only four out of eight cores were in use.

The work described in this paper is part of an ongoing project to develop a domain-specific optimisation framework for industrial visual effects. Metadata underpins our approach to performance optimisation, retaining useful information that is lost or obscured within the program. We are presently exploring a range of increasingly complex visual effects in order to identify new metadata and to broaden the applicability of our collection system. In particular, we are investigating the limits of algorithmic skeletons as a means of describing the behaviour of industrial visual effects algorithms. We are also interested in identifying domain-specific metadata which may enable targeted optimisations in the visual effects field or for subsets of the algorithms within.

References

1. T.J. Ashby, A.D. Kennedy, and M.F.P. O'Boyle. Cross component optimisation in a high level category-based language. In *10th International Euro-Par Conference on Parallel Processing*, volume 3149 of *LNCS*, pages 654–661, 2004.
2. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.
3. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, September 2004.
4. Olav Beckmann, Alastair Houghton, Michael Mellor, and Paul Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *International Seminar on Domain-Specific Program Generation, Dagstuhl, Germany*, volume 3016 of *LNCS*, pages 291–306. Springer-Verlag, March 2003.
5. Olav Beckmann, Paul Kelly, and Peter Liniker. Delayed evaluation, self-optimising software components as a programming model. In *8th International Euro-Par Conference on Parallel Processing, Paderborn, Germany*, volume 2400 of *LNCS*, pages 323–342. Springer-Verlag, 2002.
6. Olav Beckmann and Paul H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In *LCR '98: Selected Papers from the 4th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511 of *LNCS*, pages 123–138. Springer-Verlag, 1998.

7. A. Benoit, M. Cole, J. Hillston, and S. Gilmore. Flexible skeletal programming with eSkel. In *10th International Euro-Par Conference on Parallel Processing*, volume 3648 of *LNCS*, pages 761–770. Springer-Verlag, 2005.
8. Lubomir Bourdev and Hailin Jin. Generic Image Library design guide. `http://opensource.adobe.com/gil/gil_design_guide.pdf`, December 2006.
9. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In *ACM Int. Conf. on Supercomputing (ICS'05), Boston, Massachusetts*, June 2005.
10. Krzysztof Czarnecki, Ulrich W. Eisenecker, Robert Glück, David Vandevoorde, and Todd L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, volume 1766 of *LNCS*, pages 25–39, London, UK, 2000. Springer-Verlag.
11. Nathalie Furmento, Anthony Mayer, Stephen McGough, Steven Newhouse, Tony Field, and John Darlington. Optimisation of component-based applications within a grid environment. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, New York, NY, USA, 2001. ACM Press.
12. Paul Kelly, Olav Beckmann, A. J. Field, and Scott Baden. THEMIS: Component dependence metadata in adaptive parallel computations. *Parallel Processing Letters*, 11(4), 2001.
13. Christian Lengauer. Loop parallelization in the polytope model. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 398–416, London, UK, 1993. Springer-Verlag.
14. Cristina Nicolescu and Pieter Jonker. EASY PIPE: An "easy to use" parallel image processing environment based on algorithmic skeletons. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 114, Washington, DC, USA, 2001. IEEE Computer Society.
15. Karen Osmond, Olav Beckmann, Anthony J. Field, and Paul H. J. Kelly. A domain-specific interpreter for parallelizing a large mixed-language visualisation application. In *18th International Workshop on Languages and Compilers for Parallel Computing*, volume 2958 of *LNCS*, pages 347–361. Springer-Verlag, 2005.
16. Sebastian Pop, Georges-André Silber, Albert Cohen, Cédric Bastoul, Sylvain Girbal, and Nicolas Vasilache. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *GNU Compilers Collection Developers Summit, Ottawa, Canada*, 2006.
17. Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing.* Springer-Verlag, London, UK, 2003.
18. Francis P. Russell, Michael R. Mellor, Paul H. J. Kelly, and Olav Beckmann. An active linear algebra library using delayed evaluation and runtime code generation. In *Library-Centric Software Design LCSD'06*, 2006.
19. M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. In *Proceedings of the Joint Modular Languages Conference (JMLC'03), Lecture Notes in Computer Science*, volume 2789, pages 214–223, Aug 2003.
20. Yonghong Song, Rong Xu, Cheng Wang, and Zhiyuan Li. Data locality enhancement by memory reduction. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 50–64. ACM Press, 2001.
21. Antonio De Stefano, Bill Collis, and Paul White. Synthesising and reducing film grain. *Journal of Visual Communication and Image Representation*, 17(1):163–182.
22. Todd L. Veldhuizen and Dennis Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*, Philadelphia, PA, USA, 1998. SIAM.