High-Performance SIMT Code Generation in an Active Visual Effects Library

Jay L. T. Cornwall
Department of Computing,
Imperial College London, UK
jlc01@doc.ic.ac.uk

Lee Howes
Department of Computing,
Imperial College London, UK
lwh01@doc.ic.ac.uk

Paul H. J. Kelly
Department of Computing,
Imperial College London, UK
phjk@doc.ic.ac.uk

Phil Parsonage
The Foundry, UK
phil@thefoundry.co.uk

Bruno Nicoletti
The Foundry, UK
bruno@thefoundry.co.uk

ABSTRACT

SIMT (Single-Instruction Multiple-Thread) is an emerging programming paradigm for high-performance computational accelerators, pioneered in current and next generation GPUs and hybrid CPUs. We present a domain-specific active-library supported approach to SIMT code generation and optimisation in the field of visual effects. Our approach uses high-level metadata and runtime context to guide and to ensure the correctness of optimisation-driven code transformations and to implement runtime-context-sensitive optimisations. Our advanced optimisations require no analysis of the original C++ kernel code and deliver 1.3x to 6.6x speedups over syntax-directed translation on GeForce 8800 GTX and GTX 260 GPUs with two commercial visual effects.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]; I.3.1 [**Hardware Architecture**]: Graphics processors; I.4.0 [**General**]: Image processing software

General Terms

Algorithms, Design, Performance

1. INTRODUCTION

The Single-Instruction Multiple-Thread (SIMT) programming model [9] is promoted in CUDA and OpenCL for programming graphics processors. Automatic parallelisation and optimisation of arbitrary affine loop nests for a SIMT architecture is complex but carries large performance benefits [2]. In this paper we demonstrate a domain-specific approach to SIMT parallelisation and optimisation that is both simpler to implement and capable of more advanced optimisations. We use high-level metadata to communicate data dependence patterns from visual effects algorithms directly to a source-to-source compiler, built upon the ROSE [13] infrastructure. These patterns are typically difficult to extract from C implementations of the algorithms, creating a major barrier to a domain-agnostic compiler approach. Our code generation techniques benefit from the runtime context captured by an active library [5], enabling optimisations that are specialised to sequences of kernels within a visual effects DAG. We present a case study of the performance of two commercial visual effects algorithms, shown in Figure 1, built from compositions of nine primitive operations of varied complexity and data access requirements.

In previous work [4] we described a framework for constructing image processing algorithms from DAGs of primitive operations and generating optimised code for a CPU through cross-component loop and kernel transformations. We revisit and extend this framework in Section 2 to support more complex primitives, but the fundamental principles involved in our earlier CPU optimisation work remain unchanged for SIMT devices: a testament to the robustness of a domain-specific metadata-supported approach.

The main contributions of this paper are:

- **A metadata-enhanced data-parallel model for visual effects**. Section 2 outlines a C++ library for writing reusable primitives and effects, encapsulating static and dynamic algorithm metadata to assist code generators in deploying loop and kernel optimisations.
- **Domain-specific optimisations for SIMT architectures**. Section 3 identifies the key performance challenges in writing programs for a SIMT device. Section 4 describes a set of optimisations targeted at visual effects algorithms on SIMT devices. Section 5 evaluates their effectiveness on two commercial effects and identifies those that survive the transition to a previously untested generation of GPUs, and thus those more likely to apply to a wider class of SIMT devices.

2. A METADATA-ENHANCED VISUAL EFFECTS FRAMEWORK

In this section we describe the programming framework in full. A visual effect is expressed as a DAG of primitive operations connected by intermediate images. Images are not fixed in size and, indeed, do not exist at the construction level. They represent data sets which the back-end code generators may instantiate and fill with real image

CF'09, May 18–20, 2009, Ischia, Italy.

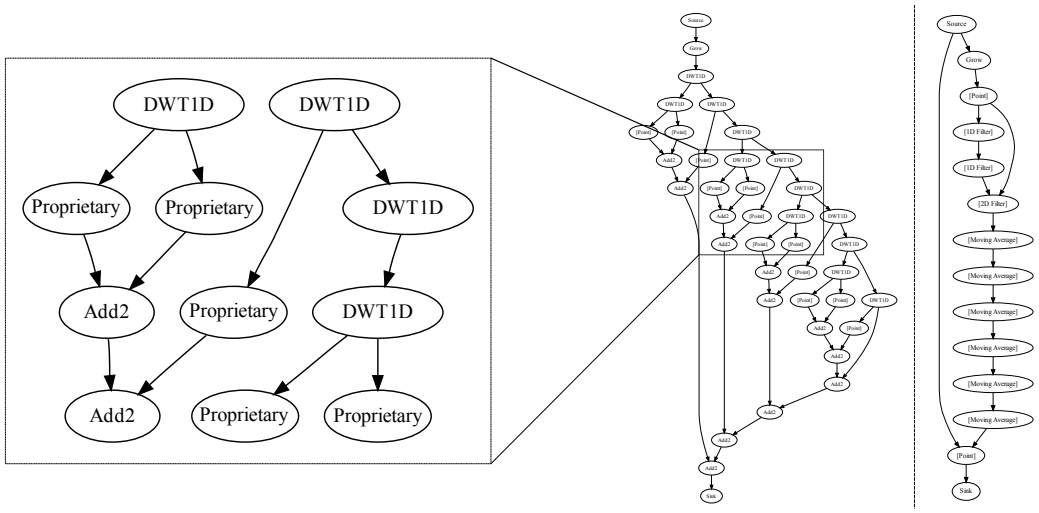

Figure 1: DAGs of primitive operations for wavelet-based degraining (left) and diffusion filtering (right) commercial visual effects. Each node represents a primitive operation and each edge carries image data.

data at runtime; or, through optimisations such as array contraction [16], may not. The primitives permitted within a DAG process whole images which are passed through a dependence-preserving serialisation of the DAG. A complete construction for a commercial wavelet-based degraining [17] effect is shown in Listing 1. The C++ code calls methods on primitive objects with input and output image handles to connect them into a DAG. Delayed evaluation makes this process flexible; in this example we make use of recursion to generate the large DAG shown on the left of Figure 1 from a repeating pattern of primitives.

```
Image DeGrainRecurse(Image input, int level = 0) {
  Image HY, LY, HH, HL, LH, LL, HHP, HLP, LHP, LLP;
  Image pSum1, pSum2, output;

  DWT1D hDWT(eHorizontal, 1 << level);
  DWT1D vDWT(eVertical, 1 << level);
  hDWT(input, HY, LY);
  vDWT(HY, HH, LH);
  vDWT(LY, LH, LL);

  Proprietary prop;
  prop(HH, HHP);
  prop(LH, LHP);
  prop(HL, HLP);

  Sum sum;
  sum(HHP, LHP, pSum1);
  sum(HLP, pSum1, pSum2);

  /* Go to the next level of recursion. */
  LLP = (level<3) ? DeGrainRecurse(LL, level+1) : LL;

  sum(pSum2, LLP, output);
  return output;
}
```

Listing 1: The recursive degrain algorithm in C++. Indexed functors are chained with images to form a DAG. Primitives do not execute as they are called but are recorded through delayed evaluation.

Primitives are data parallel expressions of algorithms with constrained global memory access and high-level metadata. They are implemented as C++ classes, analogous to function objects, providing a method which is called repeatedly to process all of the data elements within a set of images. Special member objects, which we call *indexers*, simplify and constrain reads and writes to global image data. We call the complete construction an *indexed functor*. Listing 2 shows an indexed functor which implements the one-dimensional discrete wavelet transform primitive, which is used in the degraining visual effect. An indexed functor may be embarrassingly parallel, evident in this case by the *eParallel* template metadata, or it may have parallelism along a single axis – with a loop-carried dependence along the perpendicular axis – by specifying an additional template parameter. Indexed functors with a loop-carried dependence are called moving averages, although they do not necessarily compute a mean. This specification of parallelism, expressed as class metadata, is sufficient to optimise the two commercial visual effects studied in this paper for SIMD and SIMT devices.

```
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput, eComponent, e1D> Input;
  Indexer<eOutput, eComponent> HighOutput;
  Indexer<eOutput, eComponent> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(IndexerAxis axis, IndexerRadius radius)
    : Input(axis, radius) {}

  void Kernel() {
    float centre = Input();
    float high = (centre - (Input(Input.Radius)
      + Input(-Input.Radius)) * 0.5f) * 0.5f;

    HighOutput() = high;
    LowOutput() = centre - high;
  }
};
```

Listing 2: The one-dimensional discrete wavelet transform as a C++ indexed functor implementation. It is valid, compilable code and operates in the runtime-parameterised horizontal or vertical axis. Static and dynamic metadata is underlined. Dynamic metadata is provided by the client at runtime during object construction.

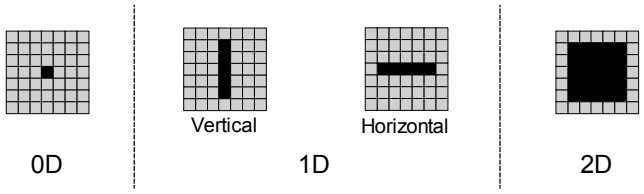

Figure 2: Indexers for global memory access with local offsets in different degrees of freedom and spatial orientations.

An indexed functor may contain small amounts of read-only (*const*) member data but may only access large images through indexers. An indexer's purpose is twofold: it centralises accesses over the region of an image that a single thread is responsible for and constrains the spatial freedom of local offsets. In Listing 2 the *Input(Input.Radius)* and *HighOutput()* component accesses, for example, do not refer to an (x,y) location in the image. Rather, they are centred over some point in the image and it is up to the code generator to iterate the kernel over the full range of coordinates. The *Input(Input.Radius)* access is parameterised with an offset which is used to locally shift the data access along some axis of freedom. Static template metadata in the indexer's declaration specifies its number of degrees of freedom: e.g. the *e1D* template parameter in Listing 2. Indexers are additionally constructed with dynamic information about the axis of freedom and limits upon the size of permissible local offsets. The different kinds of indexers supported within the framework are shown in Figure 2. *Input* is a 1D indexer so by parameterising accesses with an offset it can read to the left and right or above and below the centred coordinate depending on the axis with which it was constructed at runtime. This makes it easy to combine horizontal and vertical filter primitives into a single indexed functor implementation. Finally, an indexer may provide access to more than one data element from different image planes – red, green, blue, etc. – if they are needed together. Those in Listing 2 access single planes of input/output images at a time, evident by their static *eComponent* metadata. It is more efficient to process a single plane from all images at a time, in terms of kernel complexity and working set size, but some algorithms are dependent on chromatic information.

In summary, the complete set of metadata consists of:

- **DAG construction from indexed functors and images**. Static expression of the visual effect construction from primitives, such as that shown in Listing 1, captured through delayed evaluation.
- **Indexed functor dependence**. Static choice of embarrassingly parallel or loop-carried dependence along an axis, with dynamic selection of the axis in the latter case. See underlined features of Listing 2.
- **Indexer access patterns**. Static choice of per- component or per-pixel granularity and 0D, 1D or 2D spatial offset freedom. Spatial offsets are bounded by dynamic radii. See underlined features of Listing 2.

Once an effect has been constructed it is passed through an optimising code generator, embedded in the framework library, for a computational device of interest. An example of a simpler kernel optimised for a SIMT device is shown in Listing 3. Poor memory access patterns in the indexed functor are identified from its metadata and rectified through a staged realignment optimisation, described in detail in Section 4.4. Writing and maintaining optimised kernels like these by hand is very difficult and error-prone, and requires consideration of vastly different performance characteristics for each device. Given the metadata collected so far, our contribution is an optimising SIMT code generation backend for this framework. The remainder of the paper shows how this metadata is used to deploy advanced kernel optimisations and construct high-performance SIMT kernels for arbitrary indexed functors, with no analysis of the C++ kernel or construction code.

```
__global__ void DWT1D_HM(float *Input,int Input_YStr,
  int Input_PStr,int Input_Radius,int Input_Misalign,
  float *HighOutput,float *LowOutput,int _dyStr,
  int _dpStr,int _xElems,int _yElems)
{
  __shared__ extern float _sMem[];
  const int _blkX=__mul24(blockIdx.x,blockDim.x);
  const int _blkY=__mul24(blockIdx.y,blockDim.y);
  const int _thrX=_blkX+threadIdx.x;
  const int _thrY=_blkY+threadIdx.y;

  Stage1DRealign(&Input[_thrY*Input_YStr+_blkX
    -Input_Radius],_sMem,blockDim.x+
    __mul24(Input_Radius,2),_xElems+
    __mul24(2,Input_Radius)-_blkX,Input_Misalign);

  __syncthreads();

  if(_thrX<_xElems&&_thrY<_yElems) {
    float centre=_sMem[threadIdx.x+Input_Radius];
    float high=((centre-((_sMem[threadIdx.x
      +Input_Radius+Input_Radius]+_sMem[threadIdx.x
      +Input_Radius+(-Input_Radius)])*0.5F))*0.5F);
    HighOutput[_thrY*_dyStr+_thrX]=high;
    LowOutput[_thrY*_dyStr+_thrX]=(centre-high);
  }
}
__device__ inline void Stage1DRealign(float *from,
  float *to,int nElems,int guardX,int amount)
{
  for(int i = (threadIdx.x+amount) & (blockDim.x-1);
    i<nElems && i<guardX; i+=blockDim.x)
  {
    to[i] = from[i];
  }
}
```

Listing 3: An automatically generated, optimised horizontal DWT kernel expressed in CUDA. Shared memory staging and realignment optimisations, described in Sections 4.1 and 4.4, have been applied to localise shared data in high-bandwidth memory and to improve the efficiency of DRAM transactions.

3. CHALLENGES IN SIMT CODE GENERATION

In our study of the SIMT model we focus on the Compute Unified Device Architecture (CUDA) [10] on NVIDIA GPUs. Of competing standards only the Open Compute Language (OpenCL) has gained significant industry support. OpenCL is based heavily upon CUDA and positioned as a hardware-independent standardisation of tools for the SIMT programming model. NVIDIA, AMD and Intel have pledged support for this standard on their GPUs and hybrid CPU/GPUs. Our research, then, should apply well to both current and next generation high-performance architectures.

CUDA is programmed with thread-centric kernels. Syntax -directed translation of a C++ indexed functor kernel is sufficient to produce a correct SIMT implementation of the algorithm. Stripwise parallelisation, as employed by scalar and SIMD code generation, or even pointwise parallelisation is sufficient to exploit the massive parallelism of SIMT architectures. However, the performance of this approach has a number of shortcomings:

- **Threads do not cooperate**. Substantial throughput gains can be made by sharing the data retrieved from global memory amongst threads. There is significant overlap in the reads issued by neighbouring threads within a stencil operation, for example. SIMT architectures with very limited hardware-managed caches – such as NVIDIA and AMD GPUs – rely on explicit cooperation through shared memory and synchronisation primitives. The hardware-managed cache-dependent C++ kernel does not have support for these features.
- **Memory accesses may not coalesce**. The global memory systems of SIMT architectures can deliver an order of magnitude higher performance when the reads and writes of groups of threads meet alignment and ordering criteria, a feature referred to as coalescing. These are similar to the alignment and blocked read-/write constraints of SIMD architectures but are distinguished by being unenforced in the programming model. Explicit realignment and coordinated staging into shared memory can increase coalescing opportunities.
- **Thread management costs may dominate**. SIMT architectures are designed to keep thousands of threads in flight with little overhead. A 1:1 mapping of threads to data elements on large data sets, however, can require levels of thread management that dominate the performance of simpler kernels. A more efficient 1:N mapping cannot be derived through syntax-directed translation alone.

Control loops that drive a kernel over the domain of a set of output images must also be mapped to the SIMT architecture. CUDA assigns a three-dimensional thread ID and a two-dimensional block ID to each thread. The range of these IDs is set before a computation is launched and determines the total number of threads that will be spawned. Mapping these IDs to different data elements allows the workload to be distributed amongst the threads. A simple 1:1 mapping of two-dimensional (x,y) pairs to linear combinations of thread and block ID is sufficient to process a dense data set. There are some caveats to this approach, however:

- **Thread block size and shape impacts performance**. Maintaining a balance between the number of threads per block and the number of blocks is important to achieve high occupancy on a SIMT architecture. Higher thread counts must be traded off against increased register and shared memory requirements. The shape of a thread block can affect the potential for spatial thread cooperation on a data set. Both the size and shape of a thread block can alter the "overhang" of edge blocks beyond the boundaries of a data set. Finally, the shape of a thread block may be constrained by data dependencies in either dimension.
- **Blocks and grids are limited in size**. Thread blocks may contain a maximum of 512 threads, while grids are limited to 65535 blocks in each dimension. Redundant memory accesses between thread blocks, where threads cannot cooperate, become significant in larger stencils where the thread block size cannot be increased to compensate. Furthermore, a reduction in threads per block may increase performance because the occupancy limit of N threads per multiprocessor might not divide evenly by the number of threads per block. Limitations in grid size make one-dimensional flattening of a two-dimensional data set, done to reduce linear index computation overheads, more difficult without 1:N mappings of threads to data elements.
- **Grids must be rectangular and dense**. A key barrier to mapping fragmented, imperfectly nested loop structures onto a SIMT architecture is that it cannot be achieved with a linear mapping of thread and block IDs to data elements. These structures typically arise during loop fusion optimisations. Such loop fragments may be simulated with multiple program invocations but the startup overheads involved become significant.

Once kernels and loops have been mapped to the SIMT architecture, all that remains is to add control and marshalling code. Large data sets must be transferred to and from device global memory by the host. Other parameters to the kernel, such as variables and small collections, may be marshalled into faster areas of memory, such as shared or constant memory. Control code manages both on- and off-device resources, kernel initialisation and thread launches according to the precomputed DAG serialisation for a visual effect.

3.1 Syntax-Directed Translation

The process of transforming C++ indexed functor kernels into CUDA kernels is now examined in more detail. CUDA kernels are written in a language that is very close to C++. Our syntax-directed translation phase walks the AST and translates unsupported constructs into equivalent CUDA statements whilst leaving the remainder untouched. Key features that must be translated include:

- **Indexer accesses**. Accesses to the C++ indexing objects must be translated into array accesses from their corresponding image base pointers. The array access is formed partly from a mapping of thread and block ID to an (x,y) coordinate pair, denoting the central location of the thread's workload. An optional local offset consisting of (x,y), in per-component indexers, or (x,y,c), in per-pixel indexers is then added to this central index to select the desired location and image plane. The (x,y) or (x,y,c) index is subsequently flattened to a linear offset via the image's row and plane stride parameters.
- **Member variables**. C++ indexed functor objects may contain user- and library-defined parameters, such as filter radii and sparse stencil coordinate lists, and state variables, such as partial sums. Accesses through *this–>* in the AST are translated into parameter accesses or, for state variables, local variable accesses.

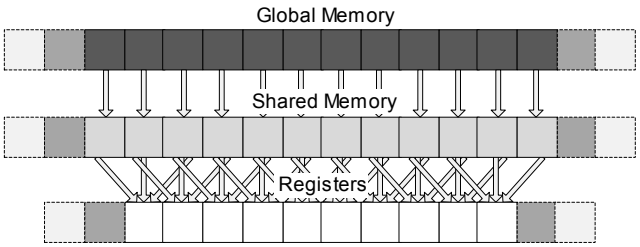

Figure 3: Coordinated staging of shared data from global memory into shared memory to localise the overlap.

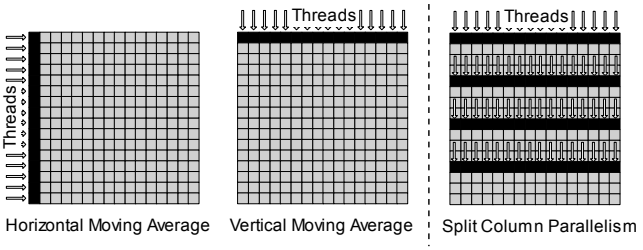

Figure 4: The contiguous access region in a vertical moving average indexed functor permits coalescing, whilst the disjoint horizontal case does not. The two configurations can be interchanged through transposition.

The goal of this phase is to produce a correct CUDA kernel implementation for the other phases to transform through various optimisations. It is used as the "unoptimised" implementation benchmarked in Section 5.

4. DOMAIN-SPECIFIC SIMT CODE OPTIMISATIONS

The result of syntax-directed translation is a correct SIMT implementation of an indexed functor. In this section we discuss the key domain-specific optimisations needed to achieve high performance on a SIMT device. Each subsection identifies a phase in our source-to-source compiler responsible for a specific optimisation.

4.1 Shared Memory Staging

Staging is a cooperative process in which threads that read overlapping regions of global memory coordinate to read each element exactly once into a faster, shared memory where the overlapped reads can benefit from higher bandwidth. The only sources of overlapped reads in our framework are the one- and two-dimensional filter indexers. Figure 3 illustrates the staging process for a 3-tap horizontal filter. There are two cases in which a filter indexer will not be staged:

- **The indexed functor is not embarrassingly parallel**. Threads do not normally share data in a moving average indexed functor because a single thread processes the data elements that would otherwise be shared in the direction of the dependence. One could construct an indexed functor where a filter indexer's axis is at 90° to the dependence axis, but this is an unlikely case and we choose not to optimise it.
- **There is insufficient space in shared memory**. Either a single large filter indexer or several filter indexers may exceed the amount of shared memory available for staging. In this case one might try to choose a subset of indexers that best fits into the available space and stage only those.

Staging is implemented in two steps. Firstly, function calls are inserted to perform a block copy of each filter indexer's access region, bounded by known limits from the runtime metadata, into a subregion of shared memory. The copy's reads and writes are distributed amongst the available threads and barrier synchronisation ensures the copy is completed before threads begin reading from shared memory. Array accesses are then redirected from the staged global memory region to shared memory. Since the shared memory contains only the subregion of interest, the (x,y) spatial offset is replaced by a local offset inside the staged block.

4.2 Indexed Functor Transposition

Global reads through an indexer in a horizontal moving average indexed functor will not coalesce. Coalescing depends upon a group of threads coordinating to read a contiguous region of memory. Dependence in the horizontal axis forbids multiple threads to run along the axis where the contiguous reads would otherwise take place. Contrast this with the vertical moving average, which achieves perfect coalescing as shown in Figure 4. Each thread issues a single read in the diagram. The combined region is discontiguous in the horizontal case, and thus not coalesced, but contiguous in the vertical case. In practice it is often faster to simply spawn one thread per pixel, each initialising its state by reading the whole filter region beneath it, and to treat the indexed functor as if it had no dependence or state at all. We use this approach in the unoptimised case as the higher thread count helped to hide the latency of uncoalesced global accesses.

It is strongly in our interests to prefer vertical moving averages over their horizontal counterparts. A simple way to achieve this is to transpose the input data sets to a horizontal moving average indexed functor, executing it with the equivalent but faster vertical implementation, and then transpose the results back. This would pay off if the total runtime of the three stages – transpose, execute, transpose – was less than the horizontal indexed functor's runtime. There is an opportunity for further optimisation here if the context of the indexed functor is known. Sequences of horizontal moving averages – e.g. a series of box blurs making up the horizontal part of a Gaussian blur approximation – may be optimised by noting the redundant double transposition at the boundaries between them. By eliminating pairs of transposes at the graph level the effect's performance can be increased.

A similar problem arises in the case of embarrassingly parallel indexed functors with vertical filter indexers. There is a trade-off between using a vertical thread block to stage the overlapped region into shared memory, suffering discontiguous column-wise global memory reads, or running the threads horizontally to achieve coalescing but sharing no data at all. In practice we find the latter is faster. However, transposing the entire indexed functor is a better option as it enables both coalescing and shared memory staging.

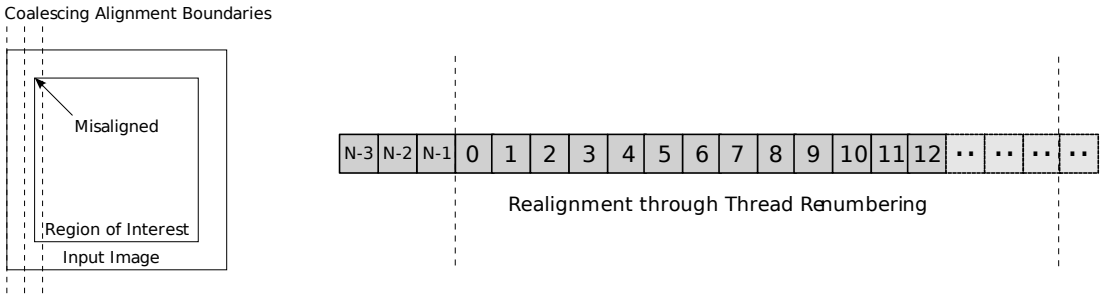

Figure 5: Global memory load misalignment occurs when the region of interest is smaller than the input image. Coalescing can be fixed by reassigning work to threads so that thread 0 is aligned to a boundary.

4.3 Split Row/Column Parallelism

Moving averages have a further performance problem. Because they constrain parallelism to a single axis only, there is frequently insufficient remaining parallelism to saturate the multiprocessors of a SIMT device. e.g. A modern GPU with 28 multiprocessors and 1024 threads per multiprocessor would need an image with 28K pixels along the parallel axis, or multiple images of a smaller size, to reach maximum occupancy. Additional parallelism along the dependence axis may be created by stopping the thread part-way and starting a new one from that point, thus breaking the data dependence chain, as illustrated on the right of Figure 4. This method creates a small overhead from state recomputation at each boundary but it multiplies with row-/ column-wise parallelism to double, triple, etc. the total available parallelism when it is required.

4.4 Realignment

The factors discussed so far have been mostly independent of the context in which the indexed functor is used. An important consideration of this context must be made in the case of coalescing. Satisfying the alignment constraints for coalescing has so far been under the implicit assumption that the base address of input images is aligned. Indeed, CUDA will guarantee that allocated regions begin on an aligned boundary for this reason. However, the base addresses of an indexed functor's regions of interest (ROI) do not necessarily coincide with the base address of the associated images. A second indexed functor may share one of the input images with a potentially larger ROI. The size of the image will satisfy the largest ROI of an indexed functor that uses it. Unfortunately, the offset of the ROI in an indexed functor is not predictably aligned. In most cases the difference between image size and ROI can be statically compared as expressions of radial terms and determined to be identical or not. The remainder can be checked at runtime with dynamic instantiations of filter radii, but we must still handle the misaligned cases.

Realignment is a staging transformation that attempts to reconcile coalescing within a thread block. A first attempt would be to redistribute work amongst the available threads so that thread 0 begins on an aligned boundary, as shown in Figure 5. This restores coalesced reads but has the unfortunate side effect of misaligning writes in the output, negating any benefit. Our solution is to use thread renumbering to first stage data into shared memory, and then use the original thread layout to process it. An example of a fully realigned kernel was shown earlier in Listing 3. Realignment cannot achieve 100% coalescing within a thread block because the first and last groups of threads are not large enough for a full coalesced read. It is most effective when the thread block is one-dimensional and large, thus minimising the constant overheads.

4.5 Thread Block Minimisation

A 1:1 mapping of data elements to threads creates considerable overheads for hardware schedulers in simpler indexed functors, particularly those composed from point indexers. An alternative is to spawn a fixed number of thread blocks and use a 1:N mapping of threads to data elements. We refer to this technique, illustrated in Figure 6, as thread block minimisation. The two-dimensional images are indexed with a single flattened index to avoid complex mapping calculations. Each thread processes a set of data elements separated by the total number of threads. Uneven mappings may result in underutilisation of some threads towards the end of the data set, but with a sufficient number of thread blocks the impact is insignificant. A summation kernel with thread block minimisation applied is shown in Listing 4.

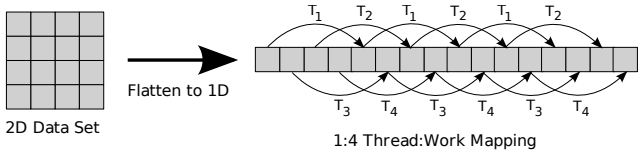

Figure 6: 1:N mapping of threads to data elements in a 2D data set to reduce thread block scheduling overheads.

```
__global__ void Add(float *Input1, float *Input2,
  float *Output, int nElems)
{
  for(int _off = __mul24(blockIdx.x, blockDim.x)
    + threadIdx.x;_off < _nElems;
    _off += __mul24(gridDim.x, blockDim.x))
  {
    Output[_off] = Input1[_off] + Input2[_off];
  }
}
```

Listing 4: A summation kernel with thread block minimisation optimisation. Scheduling overheads are reduced by assigning multiple work elements per thread. The *for* loop selects elements from the work set separated by the total number of threads, beginning from a thread-unique offset.

4.6 Thread Block Size, Shape and Count Selection

Choosing a thread block and grid configuration can drastically alter performance. These factors can even decide whether a program can run at all under the resource constraints of a given device. It is worth beginning by noting the hard limits constraining our choices of thread block size, shape and count. CUDA specifies five relevant parameters in each version of its Compute Capability (CC). T_{max} is the maximum number of threads per thread block. T_{mpm} is the maximum number of threads that can be managed concurrently on a multiprocessor. R_{max} is the total number of registers available to a multiprocessor. S_{max} is the number of bytes of shared memory per multiprocessor. Our study focuses on CC 1.0, but note that our selection techniques scale to newer versions by simply using different parameterisations of these factors. For CC 1.0 the parameters are:

$$T_{max} = 512, T_{mpm} = 768, R_{max} = 8192, S_{max} = 16000$$

For CC 1.3 the parameters are:

$$T_{max} = 512, T_{mpm} = 1024, R_{max} = 16384, S_{max} = 16000$$

T_{max} is a hard limit on the number of threads per block, but we are additionally constrained by the availability of registers (R_{max}) for those threads to use and by the shared memory requirements of a particular kernel. Focusing just on the register requirement for now, the maximum number of threads per block T_{pb} for a given kernel is related to the number of registers the kernel uses R_{pt}. The latter value can be extracted from the compiled kernel metadata. Equation 1 relates these two factors. The floor notation $\lfloor a, b \rfloor$ denotes a rounded down to the next integer multiple of b. This equation is undocumented but used in the CUDA Occupancy Calculator spreadsheet. It has value in choosing optimal thread block configurations and thus we have listed it. T_{phw} is the number of threads per half-warp, 16 for all current versions of CC.

$$T_{pb} = min(T_{max}, \left\lfloor \frac{R_{max}}{T_{phw} \times R_{pt}}, 4 \right\rfloor \times T_{phw}) \quad (1)$$

Per-kernel shared memory requirements are a little harder to formalise. Each kernel uses a fixed amount of shared memory for parameters and internal use S_{pk}, advertised in the compiled kernel metadata. The remaining shared memory consumption arises from the staging optimisations discussed in Section 4. These can be broadly classified as shared memory per-thread S_{pt} and a constant amount per-block independent of the block size S_{cpb}. Thus the maximum number of threads per block for a given kernel is additionally constrained by Equation 2.

$$S_{pk} + S_{cpb} + S_{pt} \times T_{pb} \leq S_{max} \quad (2)$$

The factors discussed so far are hard limits. In fact, performance can often be increased by substantially undercutting them. By reducing T_{pb} the shared memory S_{max} can be divided amongst multiple blocks. Additionally, CC 1.0 specifies a maximum number of threads per multiprocessor T_{mpm} of 768. A T_{pb} of 512 could not fully saturate a multiprocessor on this architecture because there is an insufficient number of free threads to process a second block on the multiprocessor. In this case reducing T_{pb} to 256 may permit three blocks to run on the multiprocessor – if shared memory and register constraints are satisfied – potentially increasing performance further than the 512 thread block could have through its more efficient intra-block communication.

Once T_{pb} has been chosen it must be virtualised into a two-dimensional thread block $T_{pbx} \times T_{pby}$. This could be a horizontal or vertical line, a square or something in-between. The shape of a thread block primarily affects the ratio of information shared amongst threads to the incommunicable but logically shared information at the boundaries of thread blocks. Maximising this ratio requires different shapes in different cases. Additionally, the shape of a thread block may be constrained by dependence in the horizontal or vertical axis.

$T_{pbx} = T_{pb}$, $T_{pby} = 1$ (a horizontal line) is required by vertical moving average indexed functors and the thread block minimised indexed functors described in Section 4.5. It is also used to maximise shared information in horizontal one-dimensional filter indexers: the shared information is T_{pb} elements and incommunicable information is only (*2 x radius*) elements. On the other hand, $T_{pbx} = 1$, $T_{pby} = T_{pb}$ (a vertical line) is required by horizontal moving average indexed functors. Note that we expect vertical one-dimensional filter indexers to have been transposed by this point, thus they would also use a horizontal thread block.

$T_{pbx} = \lfloor \sqrt{T_{pb}} \rfloor$, $T_{pby} = \lfloor \sqrt{T_{pb}} \rfloor$ is a compromise of the two approaches. It is used for two-dimensional filters and for mixes of horizontal and vertical filters. It is suboptimal in both axes but maximises the shared data ratio in the 2D filter. Rectangular variations can be used when the horizontal and vertical filter radii differ.

Once a thread block size has been chosen, the grid size $B_{pgx} \times B_{pgy}$ is fixed. In most cases this is given by $B_{pgx} = \left\lceil \frac{dodWidth}{T_{pbx}} \right\rceil$, $B_{pgy} = \left\lceil \frac{dodHeight}{T_{pby}} \right\rceil$. Thread block minimised kernels are a special case where we set $B_{pgx} = 160$, $B_{pgy} = 1$. This block count is tuned experimentally: it is small to minimise block management overheads but large enough to keep all multiprocessors occupied at reasonable thread block sizes.

5. EXPERIMENTAL RESULTS

The optimisations described in this work were tested on a CUDA CC 1.0 device: the GeForce 8800 GTX with 768MB of video RAM. As a measure of the rapid progress in this emerging field of hardware, and to assess the generality of our optimisations, we also present results for a CC 1.3 device: the GeForce GTX 260 (Core 216) with 892MB of video RAM. Throughout this section we focus solely on the execution time of CUDA kernels. In particular we do not measure host/device data transfer times because their relevance is dependent on wider application memory management issues and a hardware organisation that is rapidly evolving. All results were generated on Ubuntu Linux 8.10 64-bit systems with the 188.08 NVIDIA driver and CUDA 2.1 beta compiler.

Figure 7 presents performance results for the two commercial visual effects outlined in Figure 1 – degraining and diffusion filtering – on a CC 1.0 device. Each effect has a stock input image and parameters configured to perform a realistic image processing operation. The effect DAG is first serialised with an NP-complete algorithm to optimally minimise peak memory consumption; we are exploring faster algorithms but this completes within milliseconds. Kernel tim-

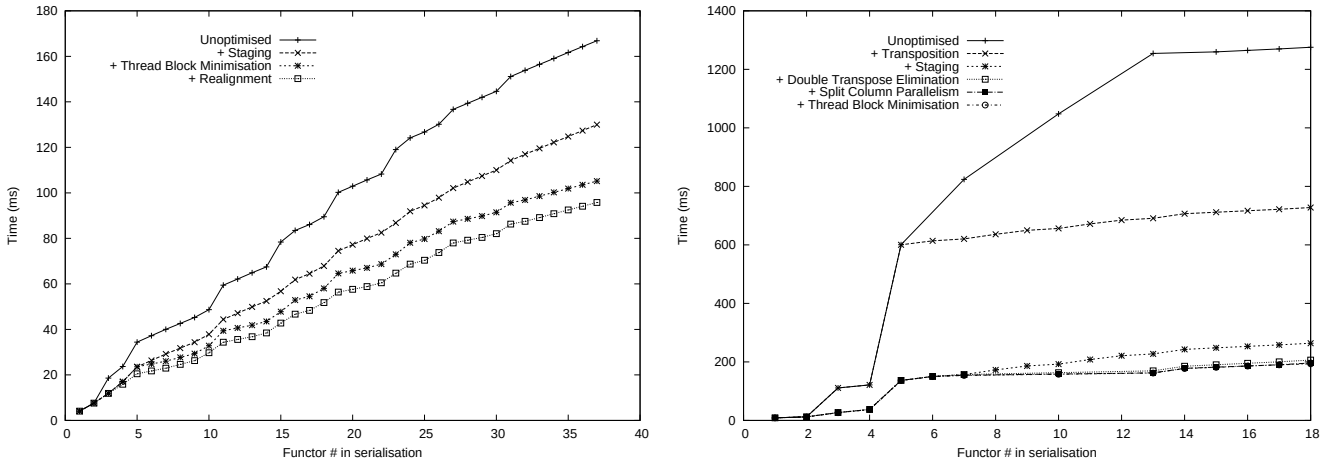

Figure 7: Degraining a 2063x1545x3 SP floating-point image (left) and diffusion filtering a 3072x2304x3 SP floating-point image (right) in CUDA on an 8800 GTX (CC 1.0). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations.

ings are accumulated as each indexed functor in the serialisation is executed. The uppermost line of each graph, with the longest execution time, represents a syntax-directed translation of the C++ indexed functors with the best thread block and grid configurations we could find. Optimisations are then applied successively, in order of greatest effect first, to produce the remaining set of lines. Where optimisations do not improve performance they are omitted from the graph and discussed in the accompanying text.

In degraining we are able to achieve an approximate 1.75x performance improvement. The shared memory staging optimisation improves data reuse in the DWT filter kernels, reducing round trips to global memory, and coordinated loads for coalescing. Thread block minimisation reduces the small but frequently incurred overheads of hardware scheduling in the effect's summation and proprietary point indexed functors. Finally, realignment improves coalescing opportunities in the DWT kernels that read subregions of larger images by introducing staging with renumbered threads. The gains made from the sum of these optimisations are modest but significant and representative of the potential gains in many simpler visual effects, which are built mainly from point-based operations and small filters. Transposition and split row/column optimisations do not apply.

Diffusion filtering sees a more dramatic improvement of approximately 6.6x. This algorithm is dominated by an intensive sparse 2D filter at position five in the serialisation, causing the steepest growth in execution time. Graph-level transposition optimisations alleviate the heavy performance penalties incurred in horizontal moving average indexed functors by enabling coalesced global reads. Shared memory staging helps to reduce the massively redundant global memory accesses in the 2D filter and improves coalescing through coordinated reads. A graph-level optimisation to remove redundant pairs of transpose kernels achieves a further small improvement. Split column parallelism in moving averages is largely ineffective due to the CC 1.0 device's relatively small multiprocessor count. The gains from thread block minimisation are the same as those in degraining, but there is only one indexed functor that benefits from it and the relative performance improvement is insignificant. The effect remains dominated by the 2D filter but its inherent memory and computational complexity justifies this.

We now move on to the CC 1.3 device to see which of our optimisations still apply and how well they perform. Figure 8 presents results again for degraining and diffusion filtering. In degraining our best efforts achieve only a 1.3x speedup. Thread block minimisation applies here as well as it did on the CC 1.0 device. The staging and realignment optimisations, however, no longer improve performance. Alignment as a requirement for coalescing was dropped in CC 1.2; accesses within a half-warp merely need to be in the same segment of memory in order to coalesce. The sequential data access pattern is likely to result in few segments being accessed and thus the overhead of realignment – staging and additional register usage – dominates and reduces the overall performance. The staging optimisation causes a small slowdown for the DWT indexed functors, which touch only three points of their 3, 5, 9 and 17 element wide access regions. Neighbouring threads will have some overlap into these regions but the amount of reuse is considerably less than in a dense filter. We would still expect a small speed-up from staging but there are evidently overheads which we have not yet identified.

In diffusion filtering the effects of relaxed coalescing rules are obvious: one- and two-dimensional filters that do not coalesce on a CC 1.0 device run 5.6x faster before any optimisations have been applied. Transposition of horizontal moving average indexed functors sees the largest improvement, as the concurrent global memory reads are always in different segments and coalescing cannot take place. Staging again has a reduced effect in improving coalescing but still makes small gains by reducing round trips to global memory. Eliminating pairs of transpositions at the graph level makes the same gains as before. Split column parallelism helps to saturate the larger number of multiprocessors on the CC 1.3 device: 28 vs 16 before. As with the CC 1.0 device, thread block minimisation has a negligible impact here. The overall performance gain remains a respectable 2.0x.

In summary, our optimisations effectively target the per-

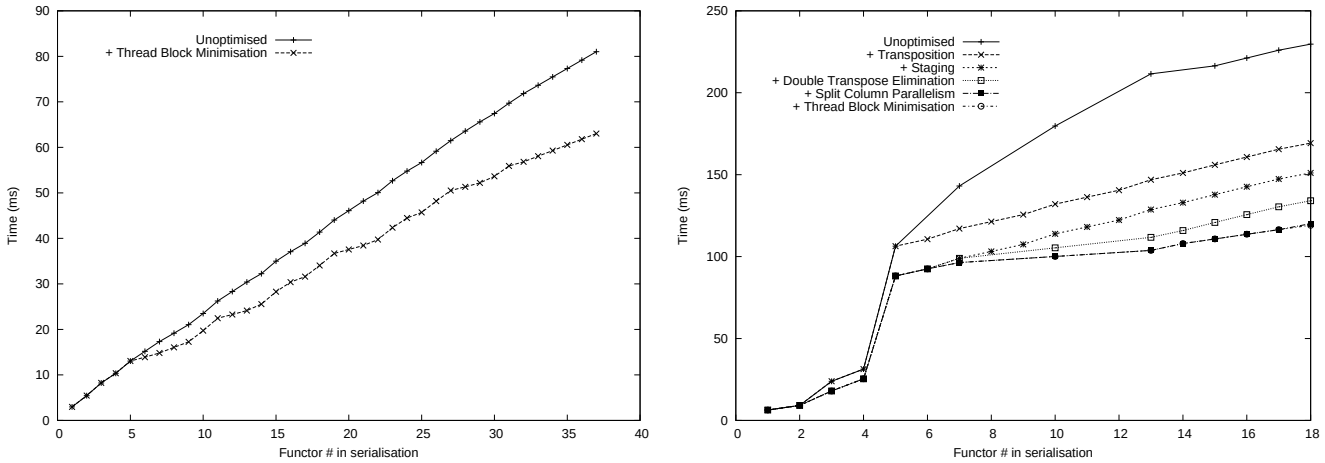

Figure 8: Degraining a 2063x1545x3 SP floating-point image (left) and diffusion filtering a 3072x2304x3 SP floating-point image (right) in CUDA on a GTX 260 (CC 1.3). Kernel execution times are cumulative across the DAG serialisation and optimisations are applied incrementally down the graph to improve performance. Gaps indicate transpose primitives that are not present in a particular composition of optimisations.

formance problems identified on the CC 1.0 device. These were primarily concerned with achieving coalesced global memory access. The CC 1.3 device has a different set of performance limiting factors – or perhaps a differently weighted set of the same factors – and more work is needed to better target our optimisations. In spite of this some of our optimisations continued to work well on the CC 1.3 device and we expect that these will be more useful to SIMT devices as a whole.

6. RELATED WORK

A domain-agnostic C++ framework with generic dependence specifications for efficient code generation on the Cell is presented in [7]. The approach is able to specify dependence more precisely than our framework but this level of detail is not yet exploited. A parallelising compiler for the SIMT architecture is described in [2] through static code analysis. Results are presented for an optimised matrix multiply kernel, exceeding the performance of a vendor-optimised library, but it has not yet been demonstrated on more complex kernels. The CUDA-Lite [18] source-to-source compiler uses source code annotations, another form of metadata, to optimise shared and global memory access in a CUDA program. The results are comparable to hand-optimised code. Many features of the optimisation space in CUDA programs, including the thread and grid choices that we touch upon, are identified in [12] and subjected to exhaustive and heuristic searches in order to better understand them.

Other notable abstract data-parallel frameworks include RapidMind [6] and PeakStream [11], two commercial toolkits with code generators for the CPU, GPU and Cell. RapidMind implements a SPMD programming model, upon which SIMT is built, and uses runtime code generation to adapt compiled programs to dynamic context. This includes cross-component loop fusions. PeakStream began as the academic Brook project [3], with similar goals, but its runtime functionality was weaker than RapidMind's before the project was acquired by Google in 2007. Brook+ is another extension to the Brook framework. The programming model is simple but requires substantial work on the programmer's part to optimise for the backend architecture [1]. OpenCL is expected to supersede much of this functionality in the near future. It fits the SIMT model [10] precisely and performs runtime code generation on AMD and NVIDIA GPUs and on Intel's Larrabee hybrid CPU/GPU.

Looking beyond the SIMT model, domain-specific frameworks for image processing have been studied for many years, driven mainly by performance demands in computer vision. In [14] the authors describe a library for shared and distributed memory CPU parallelisation of computer vision algorithms expressed in image algebra, which encompasses the functionality of our point and filter indexers. Complete effects are similarly built from primitive operations into DAGs from straight-line code. The SKIPPER project [15] pursued similar performance goals with algorithmic skeletons, upon which our theme of indexers is based, abstracting the common patterns of parallel computations found in computer vision algorithms. The authors opted for a static code analysis approach. Earlier work in [8] again built upon the idea of patterns in communication and computation as programming abstractions for algorithms. The paper discusses the problem of task scheduling, for which we use an NP-complete optimal algorithm, which they solve with heuristic-guided graph isomorphism algorithms in order to reuse subgraph architectural mappings that are known to work well.

7. CONCLUSIONS AND FURTHER WORK

In this paper an extension to a domain-specific active library was developed to tackle key SIMT architectural performance problems in commercial visual effects software. The dominating issues concerned efficient use of the global and shared memory subsystems. To alleviate these problems black-box code transformations, requiring no kernel analysis, were identified and implemented in a source-to-source compiler. High-level metadata, captured from static class decorations and runtime context through delayed evaluation, provided sufficient information to select and imple-

ment each transformation automatically. The full suite of optimisations was tested on two visual effects, donated by our industrial partners, and delivered speedups of between 1.3x and 6.6x on both our development NVIDIA GPU and a previously untested NVIDIA device with substantial architectural modifications. We showed that our optimisations are necessary for high performance and that, at least in this case study, we are able to synthesise complex data movement code automatically. A subset of optimisations survived the generational transition and we expect them to apply to a wider set of SIMT devices.

The kernel-blindness of our techniques suggests that a set of metadata-dependent rules may underpin effective CUDA code generation for arbitrary indexed functors. Certainly, the metadata identified in Section 2 and used throughout Section 4 was sufficient to achieve large speed-ups on the indexed functors considered in this paper. Node neighbours in the DAG, data dependence and memory access patterns were all used to identify appropriate optimisations for each indexed functor. Further work is needed to determine if this information alone is sufficient to support the key optimisations across all possible indexed functors. Furthermore, the scope of indexed functors requires expansion to express the complex data dependence and memory access patterns found in more advanced visual effects algorithms, such as unstructured and structured grid computations, Fourier transforms, dense and sparse linear algebra, particle simulations and Monte Carlo methods. We plan to explore these domains with single-source, high-performance code generation research on SIMD and SIMT architectures in the near future.

8. ACKNOWLEDGMENT

We would like to thank our Industrial CASE sponsors at The Foundry for contributing commercial visual effects software for this study. This work was partly funded by the EPSRC (ref EP/E002412).

9. REFERENCES

[1] AMD. Stream computing user guide. pages 50–53, October 2008.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A compiler framework for optimization of affine loop nests for GPGPUs. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 225–234, New York, NY, USA, 2008. ACM.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[4] J. L. T. Cornwall, P. H. J. Kelly, P. Parsonage, and B. Nicoletti. Explicit dependence metadata in an active visual effects library. In *LCPC*, volume 5234 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2007.

[5] K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, and T. L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, pages 25–39, London, UK, 2000. Springer-Verlag.

[6] S. du Toit and M. McCool. RapidMind: C++ meets multicore. In *Dr. Dobbs Journal*, June 2007.

[7] L. W. Howes, A. Lokhmotov, A. F. Donaldson, and P. H. Kelly. Deriving efficient data movement from decoupled access/execute specifications. In *Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC)*, volume 5409 of *Lecture Notes in Computer Science*, pages 168–182. Springer, 2009.

[8] L. H. Jamieson, E. J. Delp, C.-C. Wang, J. Li, and F. J. Weil. A software environment for parallel computer vision. *Computer*, 25(2):73–77, 1992.

[9] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[10] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[11] PeakStream. High productivity software development for multi-core processors. In *WinHEC*, 2007.

[12] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded gpu. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 195–204, New York, NY, USA, 2008. ACM.

[13] 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, 2003.

[14] F. J. Seinstra and D. Koelma. User transparency: a fully sequential programming model for efficient data parallel image processing: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(6):611–644, 2004.

[15] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Comput.*, 28(12):1685–1708, 2002.

[16] Y. Song, R. Xu, C. Wang, and Z. Li. Data locality enhancement by memory reduction. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 50–64. ACM Press, 2001.

[17] A. D. Stefano, B. Collis, and P. White. Synthesising and reducing film grain. *Journal of Visual Communication and Image Representation*, 17(1):163–182, 2005.

[18] S.-Z. Ueng, S. Baghsorkhi, M. Lathara, and W. mei Hwu. CUDA-lite: Reducing GPU programming complexity. In *LCPC*, volume 5335 of *Lecture Notes in Computer Science*. Springer, 2008.