Automatically Translating a General Purpose C++ Image Processing Library for GPUs

Jay L. T. Cornwall
Department of Computing
Imperial College London
United Kingdom
Email: jay.cornwall@imperial.ac.uk

Olav Beckmann
Department of Computing
Imperial College London
United Kingdom
Email: o.beckmann@imperial.ac.uk

Paul H. J. Kelly
Department of Computing
Imperial College London
United Kingdom
Email: p.kelly@imperial.ac.uk

Abstract— **This paper presents work-in-progress towards a C++ source-to-source translator that automatically seeks parallelisable code fragments and replaces them with code for a graphics co-processor. We report on our experience with accelerating an industrial image processing library. To increase the effectiveness of our approach, we exploit some domain-specific knowledge of the library's semantics.**

We outline the architecture of our translator and how it uses the ROSE source-to-source transformation library to overcome complexities in the C++ language. Techniques for parallel analysis and source transformation are presented in light of their uses in GPU code generation.

We conclude with results from a performance evaluation of two examples, image blending and an erosion filter, hand-translated with our parallelisation techniques. We show that our approach has potential and explain some of the remaining challenges in building an effective tool.

I. Introduction

Parallel computing, a field once dominated by supercomputers and clusters, is experiencing a surge of interest in the low cost computing mass market; not just in symmetric multicore processors, but also in heterogeneous configurations with data paths specialised for particular algorithmic structures. Multimedia instruction set extensions (SSE, AltiVec, etc.) are being augmented with parallel and vector accelerators such as graphics co-processors (GPUs), games physics engines and, for example, the IBM/Sony/Toshiba cell processor. Massive advances in their performance and flexibility are offering an increasingly attractive and widespread source of processing power to application developers.

With these advances in technology comes a heavier burden on the programmer to manage their available processing resources efficiently and to employ them effectively in problem-solving. Much of today's software is written with the CPU's serial processing paradigm in mind, limiting the usefulness of parallel devices. Although the GPU was originally intended purely for graphics applications, a growing number of promising performance results have been achieved in more general applications [1]. As we demonstrated in an earlier paper [2], recent developments, notably framebuffer objects, have increased the GPU's scope, flexibility and ease of programming. Later on in this paper we present results which illustrate the performance potential of a GPU-based solution. We also present results which show that high performance is often elusive. The barriers to uptake lie to some extent in the shortage of skilled programmers, but also in the architectural limitations of GPU designs, and in the restructuring of source code that is required.

We aim to tackle this problem by developing a tool to perform the transformation from serial processing to parallel processing on graphics hardware automatically. Our source language is C++, giving what we believe to be the broadest applicability. Language complexities, such as templates and classes, have inhibited previous attempts to analyse C++ programs effectively; we take advantage of ROSE [3], a powerful source-to-source transformation library, to assist in our analyses and to perform the bulk of our code transformations.

Generic C++ parallelisation is a very complex problem and we do not aim to solve the general problem directly. Instead we focus on the computationally-intensive libraries within an application and use domain-specific knowledge of their interfaces in order to reduce the problem space. This approach is very promising for the image processing library presented in Section II, producing a feasible automated parallelisation with very little domain-specific knowledge.

The main contributions of this paper are:

- *Parallelisation with ROSE*. We employ the ROSE source-to-source transformation library in the analysis and transformation of C++ code, to detect and expose inherent parallelism in the algorithms. Section III highlights some of the challenges in analysing the semantics of a C++ program and explains briefly how we overcome them. Section IV gives an overview of our translator's design and describes in more detail how the ROSE library integrates with the analysis and transformation process.
- *Performance Evaluation*. An evaluation of the performance experienced with our translation methods is presented in Section V, indicating the practicality of automated parallelisation for a library in our problem domain. We explain some of the problems that we encountered in attaining an optimal solution and demonstrate the GPU's potential with an erosion filter. Section VI discusses a potential method to overcome the large overheads encountered in data transfer, and shows how the library paradigm greatly simplifies this problem.

1-4244-0054-6/06/$20.00 ©2006 IEEE

II. Problem Domain

This work arises from a collaboration with a visual effects software company, The Foundry[1], aiming to accelerate a large range of image processing applications by translating much of their C++ class library to run on GPUs found in commodity PCs. This is an interesting and challenging exercise, based on industrial code "captured from the wild" that uses much of the power of C++.

Our focus is on a small part of the library's functionality, called image blending. This feature merges two images into a single image, with a user-selectable combination method. The image data is an array of floating-point elements in memory, and we treat it as such in our translations. No special status is given to the image data when performing operations on the GPU, and we handle it in the same way as we would numeric data.

On the other hand, the image data structures do lend themselves well to the GPU architecture. The basic data type, single precision IEEE floating-point, can be represented and manipulated precisely in the GPU pipeline. Structural grouping of the components into RGB objects maps well to the GPU's vector processing ability, enabling us to pack the data for simultaneous computation by vector instructions.

The library's heavy use of advanced C++ features, such as classes and templates, exercises the capabilities of the ROSE library. These structural features are typical of modern software and present a fresh challenge to program analysis. To help overcome these problems, we have asserted semantic properties that can be derived from the library's interface documentation.

Our main assumption is that pointer aliasing does not occur in the input parameters. The library accepts pointers as parameters to image data held in memory. An implicit assumption in the logic of the library's features is that these areas of image data do not overlap. In cases where this is not true, the library produces results that are not useful; more importantly, loop-carried dependencies arise in the code and greatly obstruct the parallelisation process. We make the assumption that pointers do not refer to overlapping blocks of memory.

This constraint can be enforced at the library interface in order to guarantee that all implementations will satisfy it. By making this constraint explicit, which was already assumed in the library's design, we provide valuable additional information to our parallelisation analyses.

III. Challenges to Automatic Parallelisation

It is worth noting some of the challenges that this library presents to automated parallelisation. Listing 1 demonstrates how the key data structure, an array of floating-point image components, is obscured by several layers of abstraction. The immediate typedef in the class RGBColorF presents no great problem; more troubling is the class's use as a template parameter to the image class FnTexture, which subsequently refers to the array as a single pointer with no bounds information. Extensive analysis is needed to rescue the semantics of the library code operating on these classes.

```
class RGBColorF {
public:
  typedef float Component;
  Component r,g,b;
   ...
};
...
template <class SrcPix, class SrcAr>
class FnTexture {
   ...
  SrcPix *_data;
   ...
};
```

Listing 1. Obscured floating-point image array.

ROSE offers several facilities to ease this analysis. In the parsing phase, templated classes are expanded with all of the implementations that the program will use, each appearing as a separate structure in the AST. We are able to directly link the data pointer to the base type RGBColorF through a simple call to the member variable's AST node. ROSE provides an iterative method to analyse the members of the RGBColorF class, so we can determine that data points to an array of three-float components.

The excerpt in Listing 2 highlights a loop structure which differs from the conventional two-dimensional nested loop. It is designed to support image formats with padding, stride and other awkward features. The horizontal iteration is implemented in a while loop, traversing a predefined span of pixels with a pointer increment supplied by the image class. This layout precludes template-matching approaches to parallel analysis, demanding instead a pointer and data-flow analysis.

```
for(int y = ystart; y < yend; ++y) {
  if(this->isInterrupted()) {
    break;
  }

  SrcPix *dPix = dstImg->DstPixelAddr(xmin,y);

  for(int x = xmin; x < xmax;) {
    int span = xmax - x;
    int inc;

    SrcPix *sPix = srcImg->
      GetXSpanClampedFallback(x,y,inc,span);
    x += span;

    while(span--) {
      ...
    }
  }
}
```

Listing 2. Unconventional loop structure.

The ROSE library provides several built-in data-flow analyses and frameworks on which to build custom analyses. Our array recovery algorithm (see Section IV-E), for example, uses ROSE's control-flow analysis to trace paths through the loops and reconstruct array indices from pointer adjustments. We use ROSE's AST manipulation facilities to restructure the loops into a perfect nest, while retaining the ability to produce minimally-changed source code back from the AST.

[1]www.thefoundry.co.uk

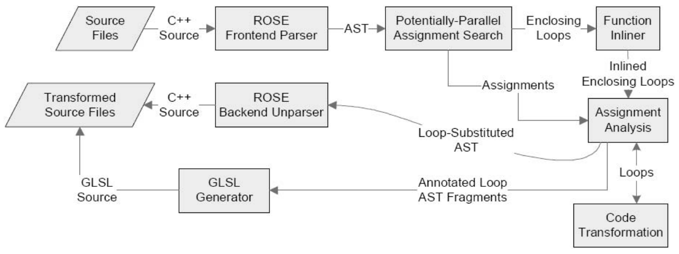

Fig. 1. High-level structure and data-flow of the translator.

Listing 3 demonstrates a subtler problem, related to GPU performance. The call to this merge function occurs on a per-pixel basis deep inside the loops shown in the previous example. Branch prediction minimises the penalty incurred when this algorithm runs on the CPU, but graphics hardware cannot employ this technique when a branch diverges within vector-sized data units; it must execute both branches regardless. It is in our interest, therefore, to lift this conditional switch to a place outside the loop using the ROSE library.

```
template <class SrcPix, class SrcAr>
class FnColorBlend {
public:
  ...
  static SrcAr merge(SrcAr &a, SrcAr &b,
    ColorBlendType t, float mix)
  {
    switch(t) {
      case eBlendHue: return blendHue(a,b);
      case eBlendLighten: return blendLighten(a,b);
      ...
    }
  }
};
```

Listing 3. Conditional switch nested deep inside a loop.

IV. Translator Architecture

In this section we present the source-to-source C++ translator. Figure 3 provides an overview of the translator's structure and the data flow between its components. The ROSE library provides two of the main components – conversion from source code to AST and back – and much of the supplementary functionality for the other components. One of ROSE's most valuable features is that comments and the layout of input source code are largely preserved in the output. This is a desirable quality where coding standards must be enforced.

A. Overview of the Translation Process

The first stage of our translation process is a search for potentially-parallel assignments (PPAs). We walk the AST and record the set of assignment statements that might be carried out in parallel: those which assign to arrays or to pointers and which are enclosed within loops. Enclosing loops are defined as those loops whose induction variables affect the offset of the assignment into the array. We record the enclosing loops with each assignment to define the scope of code that must be replicated on the GPU. At this stage we aim to find potentially-parallel loops (PPLs), which are then the focus of inlining and other transformations.

Inlining is applied to the bodies of each PPL as a precursor to analysis. This eliminates the need to perform interprocedural analyses and localises statements to the loop body. Further analysis of the set of PPAs and enclosing PPLs can then begin. We aim to confirm parallelism and to determine parameters which define the parallel assignments: contiguity, lower and upper array assignment bounds, input and output array sets, etc. This information decorates the AST for retrieval at a later stage.

We apply a range of code transformations throughout analysis to expose parallelism and to hoist undesirable features out of the loop. Switch statements, for example, are lifted through loop unswitching to avoid the penalties of branching on the GPU. These transformations are applied on-demand in response to patterns recognised by the assignment analysis. A switch statement inside a loop, for example, will trigger loop unswitching and initiate a reanalysis of the generated loops.

Once analysis is complete, the annotated AST fragments for each confirmed parallelisable loop are fed to the OpenGL Shading Language (GLSL) generation stage. The AST is walked for each loop and equivalent code for the GPU is produced. Annotations of the AST fragments are used to generate small C++ stub functions which marshall the input and output data and execute the GLSL program. These functions are substituted into the original AST, replacing each parallelisable loop.

The GLSL program is encapsulated in a C++ source file which can be added to the user's project and compiled as normal. A run-time system is provided in a linkable library, controlling the GPU's memory management, program compilation and execution. This component greatly dictates the performance of the resulting code and can be upgraded independently of the static compilation process. An example of a GLSL program, for one mode of the image blending library, is shown in Listing 4.

```
uniform samplerRect src1, src2;

void main() {
  vec3 src1_0_0 =
    textureRect(src1, gl_TexCoord[0].st).rgb;
  vec3 src2_0_0 =
    textureRect(src2, gl_TexCoord[0].st).rgb;

  gl_FragColor = clamp(src1_0_0+src2_0_0, 0.0, 1.0);
}
```

Listing 4. A GLSL program for one mode of image blending.

We now consider the most interesting aspects of the translator in detail.

B. Potentially-Parallel Assignment Search

Implemented as a complete traversal of the AST, this stage generates a set of assignments with their surrounding loops and decorates each assignment with information about the array involved. Our run-time system supports assignments of up to four floating-point elements at a time, reflecting the vector size of the GPU; operations upon more elements would require a segmentation algorithm to split them into vector-sized chunks.

AST decorations record the number of floating-point variables per array element; just one in the case of a float[] array and potentially more where arrays of classes are used.

We can directly link a pointer or array used in an assignment to its base type with ROSE. When this base type is a class, we iterate over the members of the class and count the number of floating-point variables; other member types cause the assignment to be marked as unparallel, because they introduce padding into the array. In theory, we might handle this padding in the memory unmarshalling stage, but we choose not to at this point until we have addressed performance concerns.

C. Loop Restructuring

Our need to produce parallel programs from multidimensional loop nests drives a desire for perfectly-nested loops with easily derivable bounds. This is reflected in the loop restructuring stage of our translator, which aims to transform an arbitrary loop nest into perfectly-nested *for* loops. In this form we can simply copy the loop bodies, with appropriate syntactic changes, to produce GPU algorithms. Listing 2 shows an example of the unfortunate loop nest that appears in the problem domain library.

We currently approach perfect-nesting with a limited set of methods. ROSE's inlining feature serves to remove the three function calls in the outer loop bodies. The first call is replaced by a conditional that always fails; our translator employs dead code removal to eliminate this statement. The second and third calls collapse into local variable declarations and our array recovery algorithm (see Section IV-E) removes them. We are left with the code shown in Listing 5.

```
for(int y = ystart; y < yend; ++y) {
//if(false) {
//   break;
//}

//SrcPix *dPix = &dstImg->_data[y*dstImg->_width
//   +xmin];

  for(int x = xmin; x < xmax;) {
    int span = xmax - x;
    int inc;

//   SrcPix *sPix = &srcImg->_data[y*srcImg->_width
//     +x];
    x += span;

    while(span--) {
      ...
    }
  }
}
```

Listing 5. The core loops after inlining and dead code removal.

To complete the restructuring process, the translator must consolidate the innermost *while* loop into the second *for* loop. It achieves this by first rewriting the *while* loop as a *for* loop; the loop test dictates the conditional and update statements, and the initialiser is a derived induction variable of the outer loop. The inner loop is then reversed, noting that there are no loop-carried dependencies, and merged by reducing the outer loop's step. We obtain a perfect nest of *for* loops as a result.

D. Loop Parallelisation

Once a set of PPAs has been obtained, we undertake detailed analyses of these assignments and their surrounding code to produce a set of parallelisable assignments. This stage is interwoven with code transformations that aim to increase the parallelism of the assignments, by moving invariant statements out of the loop body. We currently support only a limited subset of transformations; we plan to build on these as new code examples pose different challenges to parallelism.

Listing 6 shows a fragment of this process, handling assignment statements within the innermost loop. Prior to this stage we assume that inlining, loop restructuring (see Section IV-C) and array recovery (see Section IV-E) have taken place and that a perfectly-nested loop is the result. If this condition is not met, we abandon the parallelisation.

```
For each CFA path through the inner loop
  For each statement in the path
    Switch(type of statement)
      Case 'assignment'
        If LHS is not an array
          If LHS declared outside loop scope
            Attempt to hoist assignment
            If hoist failed
              Abandon parallelisation
          Else
            If LHS type is not 'float' or 'int'
              Abandon parallelisation
          EndIf
        Else
          Record output array and index expression
          Record arrays and input variables in RHS
        EndIf
      Case ...
      Default
        Abandon parallelisation

If output arrays differ between two CFA paths
  Abandon parallelisation

If assignments are not contiguous
  Abandon parallelisation
```

Listing 6. Pseudo-code for loop parallelisation.

The algorithm considers each possible control flow path through the loop and compares the analyses of all the paths afterwards. This is necessary to ensure that, regardless of conditions, the same set of arrays is written to. There is no efficient equivalent to not writing to an output array on the GPU; instead we would have to feed the initial state of the array as input and read from it to generate the same output. We choose instead to abandon the parallelisation for performance reasons, although we may later revise this decision if the impact is found to be small in comparison to the computation time.

Assignment statements are the most interesting elements of the loop body since they are the only effects of an algorithm on the GPU. Clearly we must also consider the side-effects of other statements in the body, and these are handled conservatively; where we cannot reproduce them in the GPU algorithm or hoist them out of the loop, the parallelisation fails. The cases for these statements have been omitted from the listing for brevity, although they are largely direct translations from C++ into the C-like GLSL.

We support assignments to variables which are live only within the scope of the loop, because they can be contained entirely within the GPU algorithm. Assignments to variables that are live-out at the end of the loop are not supported, since we have no efficient mechanism of extracting them from the GPU; instead we try to hoist these out of the loop. Hoisting itself may cause problems if the variable is used inside the loop; we mark the variable as dirty and abandon any parallelisation whose assignments reference it. On reflection, we could later support a subset of these problematic loops if the dirty variables follow strict incremental patterns throughout the loop; we can emulate these with the GPU's texture coordinates.

In addition to collecting input and output arrays for marshalling purposes, we record the index expressions used in array assignments. We analyse these expressions and determine if the assignments form a contiguous block. Where this is the case, we use the same expressions to derive upper and lower bounds for the assignments to each output array.

E. Array Recovery from Pointers

A key requirement of the parallelisation algorithm described above is knowledge of the bounds and steps of array accesses. This information is crucial in establishing that assignments to an array form a contiguous block – a prerequisite for efficient GPU processing – and in defining the sizes and locations of input and output data. We derive these characteristics by performing analyses on the array index expressions and on their corresponding loop induction variables.

This method works for well-structured programs but fails for the large body of pointer-based software. In order to alleviate these problems we apply an array recovery algorithm, temporarily converting pointer dereferences into array accesses for analysis. We based our algorithm on a similar technique [4] from the digital signal processing field.

Listing 7 shows how the image processing library uses pointers to manipulate image data. Information such as assignment bounds and step is not immediately derivable from the pointer assignment statements. Conversely, the semantically-equivalent code fragment in Listing 8 directly links assignment offsets in the array to induction variables of the surrounding loops. Our algorithm performs this transformation automatically.

```
for(int y = ystart; y < yend; ++ y) {
  for(int x = xmin; x < xmax;) {
    int span = xmax - x;

    SrcPix *dPix = &dstImg->_data[y*dstImg->_width
      +xmin];
    while(span--) {
      ...
      dPix->SetClamped(bPix);
      ++ dPix;
    }
  }
}
```

Listing 7. Pointer use in the image processing library.

```
for(int y = ystart; y < yend; ++ y) {
  for(int x = xmin; x < xmax;) {
    int span = xmax - x;

    while(span--) {
      ...
      dstImg->_data[(y*dstImg->_width+xmin)+
        (xmax-x)-span].SetClamped(bPix);
    }
  }
}
```

Listing 8. After array recovery and substitution.

V. Performance Evaluation

At this early stage of development we evaluate the performance of hand-translated examples. These conform strictly to the achievable and intended output of the completed translation software, giving representative performance of automatically-translated code. We deliberately omitted optimisations that would improve performance but which would not be implemented in the short term.

Each benchmark was run on a 3.2GHz Pentium 4 (2MB Cache) with 1GB RAM. The GPU was provided by a GeForce 7800GTX 256MB (430MHz core clock, 1.2GHz memory clock) attached to a PCI Express x16 bus. Windows XP Professional with Service Pack 2, Visual C++ .NET 2003 (7.1) and Intel C++ 9.0 were used to build and run the benchmarks, with NVIDIA's 81.95 drivers supporting the GPU. In some cases the Intel compiler generated faster code, and in others the Microsoft compiler did; when we report CPU timings, we present the faster of the two.

Figure 2 shows the performance of one mode (soft-light) of our domain library's image blending feature on the CPU and, following hand-translation, on the GPU. A range of data set sizes were tested, each consisting of two square arrays of four-float components of a width indicated on the horizontal axis. Two sets of data are provided for the GPU; this accounts for the extra overheads of context creation, data capture[2] initialisation and program compilation incurred on the first run of the algorithm, and incurred partially on the first run following a change in the size of the output data sets.

This is the most computationally-intensive mode of the library and consequently the best case for the GPU. While we see disappointing results in the initial run, with the GPU consistently slower than the CPU for all sizes of input data, on subsequent runs the GPU outperforms the CPU by an amount varying between 0-120ms. Setup overheads account for a roughly constant 200ms, from the difference between the two sets of GPU timings, independent of the data set sizes.

In the worst case we see the performance shown in Figure 3. This benchmarks the simplest mode of blending (linear-dodge) and produces disappointing results on the GPU, which is slower than the CPU in all of the cases tested. A reduction in

[2]Data capture here refers to directing the results of a GPU calculation to a general-purpose area of GPU memory, from where it can be used as an input to a subsequent computation. By default, the results of GPU computations are not necessarily stored in such a general-purpose region of memory but are instead sent to a special-purpose output-only storage region.

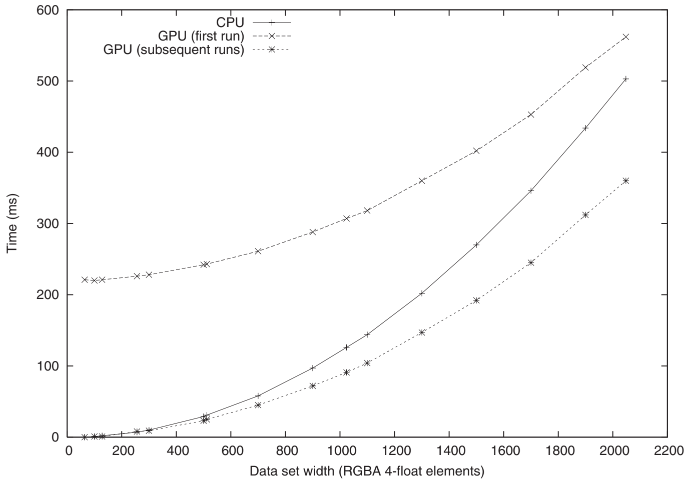

Fig. 2. Complex (soft-light) blending on two square arrays of four-element (RGBA) 32-bit floating-point values. Visual C++ 7.1 produced the fastest implementation among our compilers.

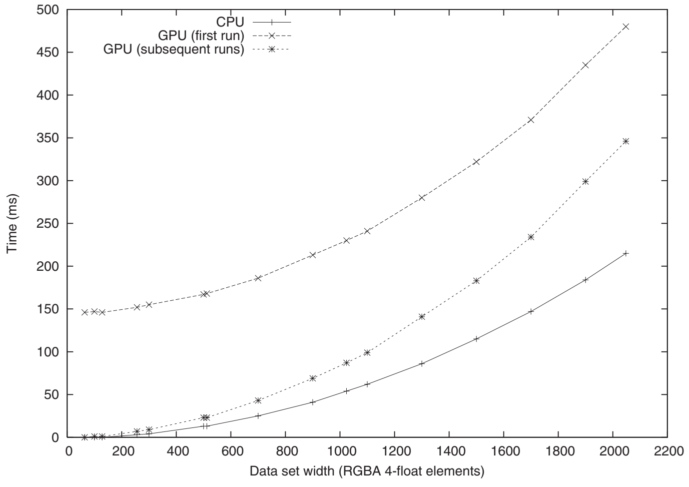

Fig. 3. Simple (linear-dodge) blending on two square arrays of four-element (RGBA) 32-bit floating-point values. Intel C++ 9.0 produced the fastest implementation among our compilers.

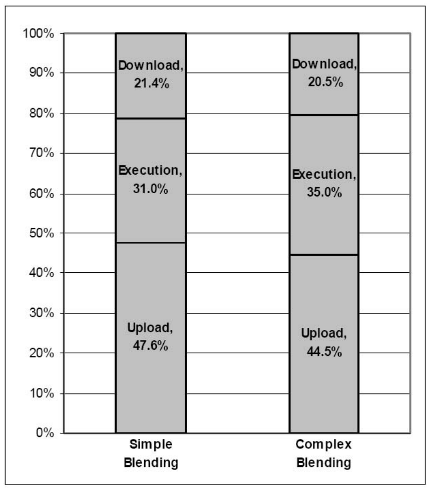

Fig. 4. Breakdown of GPU computation time for two image blending modes, on 2048x2048 data sets. Upload refers to the transfer of data to the GPU, download refers to the retrieval of results back into main memory.

the setup overheads to about 140ms reflects the smaller program compilation time. In comparison with the previous graph we see that the GPU execution time is almost unchanged; the notable difference here is that the CPU's execution time has been reduced.

In order to understand why the GPU is unaffected by this decrease in computational complexity, it is necessary to consider the structure of the execution process in graphics hardware. Figure 4 provides a breakdown of the three primary stages involved: moving input data into video memory, executing the program and moving the results back into main memory. A mere 30-35% of the total computation time is spent executing the program in this example; the rest of the time is dependent only on the sizes of the input and output data sets.

While one might expect an increase in the execution time to have perhaps a third as much impact on the total computation time, this does not fully account for the GPU's similar performance in each mode. To put this into perspective, the simpler program consists of 3 instructions while the complex program has 59 instructions; we don't see a proportional increase in the execution time. In fact, in both cases the algorithms are memory-bound; we confirmed this by reducing the core and memory clock speeds and by observing the impact on performance. Increasing the computational complexity serves largely to fill unused execution units whilst stalling for memory accesses.

Our second example shows more promise. Figure 5 demonstrates the performance of an erosion filter – a minimising convolution – on the CPU and on the GPU. Again, we employ hand-translation with the capabilities of our translator in mind. Here we see that the GPU offers large performance benefits over the CPU, even with the overheads of the first run included. The two-dimensional texture cache prefetch of the GPU allows us to incur only small penalties when accessing the image data vertically, for the kernel overlay, while the CPU appears to suffer considerably.

A breakdown comparison with the simple (linear-dodge) blending mode is offered in Figure 6. An interesting feature to note here is that the proportion of time spent uploading the input data is heavily reduced in convolution. The data behind this graph suggests that the convolution program's upload time is less than 15% of that of the blending algorithm. We might expect this figure to be 50% of the blending program, since we are dealing with only a single set of input data, but there are clearly more factors at play. This suggests that there is considerable inefficiency in streaming multiple sets of data into video memory serially. A recent paper [5] by NVIDIA offers a reason for this and proposes an alternative upload mechanism (pixel buffer objects), which we have not yet explored.

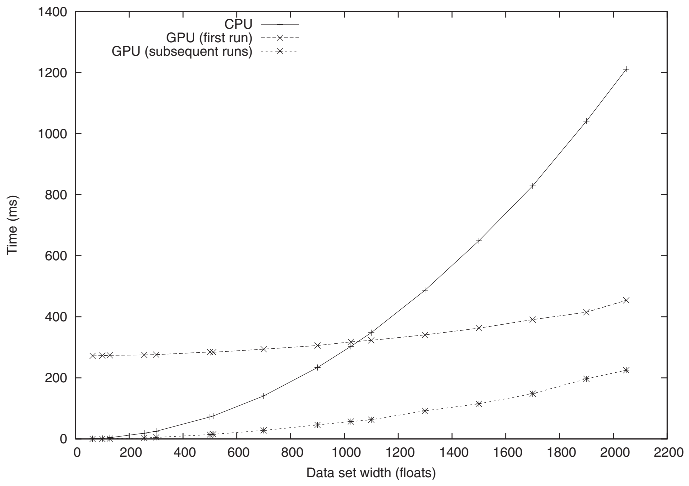

Fig. 5. An 8x8 erosion filter (convolution) on a square array of 32-bit floating-point values. Visual C++ 7.1 produced the fastest implementation among our compilers.

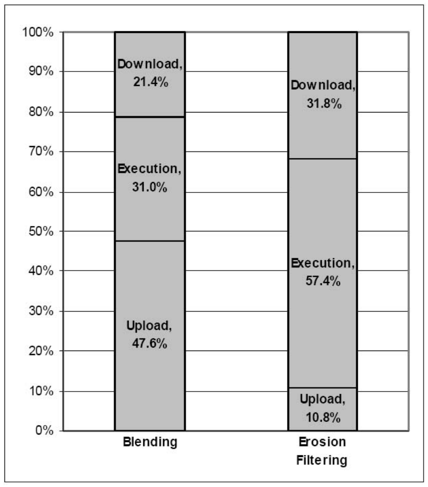

Fig. 6. Breakdown of GPU computation time for blending and erosion filtering on 2048x2048 data sets. Upload refers to the transfer of data to the GPU, download refers to the retrieval of results back into main memory.

VI. Delayed Data Retrieval

One of the most common complaints of programmers in the general-purpose GPU (GPGPU) field is that data transfer times often dominate or severely impact the total computation time, in many cases making a GPU implementation slower than the CPU. We've highlighted the degree of this problem in Figures 4 and 6, where the data transfer overheads varied between 30% and 70% of the total computation time. This inhibits the usefulness of the GPU in this domain and in many others.

In fact, we can overcome much of this overhead by avoiding the unnecessary transfer of input and output data between main memory and video memory. By localising this movement to video memory we can take advantage of the much higher bandwidth and lower latency offered by the hardware. While this may seem an obvious optimisation, determining when it is safe to hold the results of a computation in video memory is a difficult problem.

In the context of our localised loop parallelisations, the input data is uploaded just before the original loop and the results are download into arrays just afterwards. This ensures that any use of the output arrays will operate on the results of the GPU computation. In cases where the output data is not used directly, but is instead used as input to another GPU algorithm, we waste time moving the data from video memory to main memory and back again. This is a simple approach, but clearly suboptimal.

A better method is to delay the retrieval of the results of a GPU computation until they are used by the program. In cases where they are not used, and are simply fed as input to another GPU algorithm, we bypass the redundant transfer between video memory and main memory. This is similar to a delayed execution model except that there is no real benefit in delaying the program execution, rather we delay the retrieval of its results.

Implementing such a system in C++ is problematic due to the wide number of ways in which the data might be accessed; through arrays, references, through pointers and pointers-to-pointers, all with potentially overlapping areas of memory. While intrusive solutions to this problem exist, the library paradigm offers a cleaner opportunity. By encapsulating the output data structures in the library and abstracting their access through library calls, we can insert optimally-placed trigger points to initiate data retrieval from video memory.

This is an optimisation that we plan to explore in the future. Automating this process would require extensive analysis to ensure that access to the data structure couldn't be leaked without a library call, and to identify all of the points in which to insert triggers. The benefits of this work, however, could widely increase the range of algorithms suitable for GPU processing.

VII. Conclusions and Further Work

This paper offers a report on work-in-progress towards a general-purpose tool, and a methodology, for using streaming accelerators to enhance the performance of libraries in C++. We have discussed some of the code features that have proven troublesome, and we have briefly explored how semantic properties (or assumptions) of the library's API can play a part in the process. Finally, we present performance results which illustrate some of the potential of the approach and show that although very high performance is surely possible, at least soon, it is not always easy to achieve on real applications.

Our plans for this work begin with completing the automatic translation process. Whilst our translator currently supports simple parallelisations, more work is needed to automate parallelisation of the image processing library and of other complex examples. We plan to implement an interprocedural analysis to trace temporary variables through potentially-virtual function calls back to operations on data sets of interest. Greater flexibility is needed in the induction variable analysis to include affine transformations of loop control variables. Switch hoisting still needs to be implemented to generate efficient GPU code, although ROSE provides most of this functionality.

In the mid-term we aim to expand our application focus to other libraries. Intel's Computer Vision Library offers a range of computationally-intensive computer vision algorithms. These algorithms offer good potential for GPU optimisation and their performance-oriented source code presents an interesting challenge to the translator. This is a different form of parallel analysis: not through unravelling layers of structural obscurity, but through extensive loop and pointer analysis to recover the algorithm semantics.

Our long-term plans are to introduce predictive performance analysis into the translator to identify algorithms with high computation to memory access ratios and other desirable qualities for the GPU. By introducing a delayed execution mechanism we hope to fuse multiple parallel operations together to take advantage of the high bandwidth interconnects local to the GPU, avoiding slower routes back to main memory with intermediate results. We also aim to expand our back-end to support other parallel devices, such as the cell processor and FPGAs. Through careful scheduling, we can deploy parallel algorithms to multiple – perhaps heterogeneous – devices simultaneously to maximise use of the available processing resources.

VIII. Related Work

Generating code for streaming architectures like GPUs is essentially vectorisation, and is well-covered in standard textbooks such as [6]. It differs from code generation for multimedia instruction set extensions (SSE, AltiVec etc), which can be handled using sophisticated instruction selection techniques [7]. It instead resembles classical vectorisation for long-vector machines such as the Cray-1 and its successors.

The particular problems introduced by C have been tackled in many commercial compilers, and are the focus of [4] as we discussed earlier. The particular problems of (and opportunities offered by) C++ have been the focus for the development of the ROSE tool which we have used. ROSE is designed to support library-specific optimisations [3], and is motivated by the need to support scientific computing users coding with the full abstractive power of C++, while retrieving the performance attained by vectorizing and parallelizing Fortran compilers.

Exploiting the GPU for general-purpose computation is the focus of the GPGPU [1] community. Popular abstractions for general-purpose GPU programming, such as the C-like Brook [8] streaming language and the Sh [9] C++ constructs, offer simple programming interfaces with low graphics knowledge requirements.

Acknowledgments

We would like to thank The Foundry for their support for this project. This work was partly funded by the EPSRC (ref EP/C549481).

References

[1] "General-purpose computation on graphics hardware," http://www.gpgpu.org/.

[2] J. L. Cornwall, "Efficient multiple pass, multiple output algorithms on the GPU," in *2nd European Conference on Visual Media Production (CVMP 2005)*, December 2005, pp. 253–262.

[3] 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*, vol. 2789. Springer-Verlag, Aug 2003, pp. 214–223.

[4] B. Franke and M. O'Boyle, "Array recovery and high-level transformations for DSP applications," *ACM Trans. on Embedded Computing Sys.*, vol. 2, no. 2, pp. 132–162, 2003.

[5] "Fast texture downloads and readbacks using pixel buffer objects in OpenGL," http://developer.nvidia.com/object/fast texture transfers.html, 2005.

[6] M. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

[7] A. Krall and S. Lelait, "Compilation techniques for multimedia processors," *Int. J. Parallel Program.*, vol. 28, no. 4, pp. 347–361, 2000.

[8] 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.*, vol. 23, no. 3, pp. 777–786, 2004.

[9] M. D. McCool, Z. Qin, and T. S. Popa, "Shader metaprogramming," in *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2002, pp. 57–68.