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

Jay L. T. Cornwall\textsuperscript{1}, Lee Howes\textsuperscript{1}, Paul H. J. Kelly\textsuperscript{1}, Phil Parsonage\textsuperscript{2} and Bruno Nicoletti\textsuperscript{2}

\textsuperscript{1} Department of Computing, Imperial College London, UK
\textsuperscript{2} The Foundry, UK

Speaker: Lee Howes
Visual effects in film post-production

- Nuke compositing tool ([http://www.thefoundry.co.uk](http://www.thefoundry.co.uk))
- Visual effects plugins appear as nodes in complicated effect graphs
- Execution time can be many seconds per frame

(c) Heribert Raab, Softmachine. All rights reserved. Images courtesy of The Foundry.
Visual effects kernels

- Kernels – individual image processing algorithms (data parallelism)
  - Abstract computations
  - Iteration over images
  - Image memory access patterns
  - Action at each point in iteration space
  - Ordinary C++ which can be compiled and run on a CPU... slowly

```c
void DWT1D( float *input, float *highOutput, float *lowOutput, int radius ) {
    for( int y = 0; y < height; ++y ) {
        for( int x = 0; x < width; ++x ) {
            float centre = input[width*y + x];
            float high = (centre - (input[(width-radius)*y + x] + input[(width+radius)*y + x]) * 0.5f) * 0.5f;
            highOutput[width*y + x] = high;
            lowOutput[width*y + x] = centre – high;
        }
    }
};
```
Image DeGrainRecurse(Image input, int level = 0) {
    Image HY, LY, HH, HL, LL, HHP, HLP, LLP, pSum1, pSum2, output;

    DWT1D hDWT(eHorizontal, 1 << level);
    DWT1D vDWT(eVertical, 1 << level);
    hDWT(input, HY, LY);
    vDWT(HY, HH, HL);
    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;
}
Why is a new approach necessary?

• SIMD parallelism is difficult to exploit.
  • Vectorising compilers are ineffective.
  • (Only 1 out of 9 of our algorithms were vectorised by Intel C/C++ 11.0.)
  • Hand-vectorisation is difficult, error-prone and raises maintenance costs.
  • We present a related solution for this problem in an upcoming publication.

• SIMT parallelism is also difficult to exploit.
  • SIMD hardware with a parallel programming model – which requires the programmer to think in SIMD terms to get any performance.
  • Isolating sufficient parallelism (10000s of in-flight “threads”) without compromising spatial locality is challenging.
  • Data movement through the memory hierarchy requires micromanagement.
  • Hand-parallelisation is difficult, error-prone and raises maintenance costs.

• Building a compiler to do this is tricky.
  • Our innovation instead lies in metadata – to bypass tricky code analysis.
Our approach: visual effects functors

- A single-source C++ programming model.
  - Minimises maintenance costs through a **write-once** paradigm.
  - Separates the **iteration schedule** from the algorithm.
  - Carries metadata annotations. (*Underlined*, more on these in a minute.)

```cpp
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput, eChannel, e1D> Input;
  Indexer<eOutput, eChannel, e0D> HighOutput;
  Indexer<eOutput, eChannel, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius 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;
  }
};```
Our approach: visual effects functors

- Programming model that supports **focused** and **maintainable** optimisations.
  - Isolating the performance expertise to HPC developers, away from kernel authors
- An optimising **source-to-source code generator**.
  - Based on the ROSE **source-to-source** compiler framework.
  - **SIMD** and **SIMT** code generation backends.
  - A set of backend-specific **optimising code transformations**.
SIMT code is useless without optimisation

- **Shared memory staging.**
  - Localise overlapped access into **fast levels** of the memory hierarchy.
  - Each thread stages **one element** from global memory into **shared memory**.
  - Following **barrier synchronisation**, threads work from shared memory.

- **Metadata provides explicit information to make this trivial.**
  - Bypasses tricky code analysis.

![Diagram of memory hierarchy]

- **Global Memory**
- **Shared Memory**
- **Overlapped Accesses**
The kernel will be executed at each point in the iteration schedule.

```c++
void DWT( Image<float> input,
         Image<float> highOutput, Image<float> lowOutput
         int radius )
{
    for( int j = 0; j < height; ++j ) {
        for( int i = 0; i < height; ++i ) {
            float centre = Input( i, j);
            float high = (centre – (input( i, j-radius) + input( i, j-radius) * 0.5f) * 0.5f);
            highOutput( i, j ) = high;
            lowOutput( i, j ) = centre - high;
        }
    }
}

class DWT1D : public Functor<DWT1D, eParallel> {
    Indexer<eInput, eChannel, e1D> Input;
    Indexer<eOutput, eChannel, e0D> HighOutput;
    Indexer<eOutput, eChannel, e0D> LowOutput;
    mFunctorIndexers(Input, HighOutput, LowOutput);

    DWT1D(Axis axis, Radius 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;
    }
};
```
DWT – dependence metadata

- Dependence metadata is key to manipulating the iteration schedule.
  - A kernel can be **embarrassingly parallel** or have a **loop-carried dependence**.

```cpp
class DWT1D : public Functor<DWT1D, eParallel> {
  Indexer<eInput, eChannel, e1D> Input;
  Indexer<eOutput, eChannel, e0D> HighOutput;
  Indexer<eOutput, eChannel, e0D> LowOutput;
  mFunctorIndexers(Input, HighOutput, LowOutput);

  DWT1D(Axis axis, Radius 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;
  }
};
```

```cpp
void DWT( Image<float> input,
          Image<float> highOutput, Image<float> lowOutput
          int radius )
{
    for( int j = 0; j < height; ++j ) {
      for( int i = 0; i < height; ++i ) {
        float centre = Input( i, j );
        float high = (centre –
                      ( input( i, j-radius )
                      + input( i, j-radius ) * 0.5f) * 0.5f);

        highOutput( i, j ) = high;
        lowOutput( i, j ) = centre - high;
      }
    }
}
```
DWT – memory access metadata

- Memory access metadata is key to managing data movement and sharing.
- To compute one element of output, how much input does the kernel need?
- Red, green and blue together (ePixel) or one plane at a time (eChannel).
- One element (e0D), a bounded line (e1D) or a bounded rectangle (e2D).

```cpp
class DWT1D : public Functor<DWT1D, eParallel> {
    Indexer<eInput, eChannel, e1D> Input;
    Indexer<eOutput, eChannel, e0D> HighOutput;
    Indexer<eOutput, eChannel, e0D> LowOutput;
    mFunctorIndexers(Input, HighOutput, LowOutput);

    DWT1D(Axis axis, Radius 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;
    }
};
```
Box blur - dependence metadata

• An example of a kernel with a loop-carried dependence.
  • Note that this dependence is algorithmic, it is not inherent in the computation.

```cpp
void boxBlurV( Image< float > input,
               Image< float > output ) {
  for( int j = 0; j < width; ++j ) {
    float movingSum = 0.0f;
    for( int r = -radius; r < radius; ++r ) {
      movingSum += input( r, 0 );
    }
    output( i, 0 ) = movingSum;
  }
  for( int i = 1; i < height; ++i ) {
    movingSum += input( i – radius - 1, j ) +
                 input( i + radius, j );
    output( i, j ) = movingSum * multiplier;
  }
};

class BoxBlur : public Functor<BoxBlur, eMoving> {
  ...
  BoxBlur(Axis axis, int radius)
    : Functor<BoxBlur, eMoving>(axis),
      MovingSum = 0.0f,
      for(int i = -Input.Radius; i <= Input.Radius; ++ i)
        MovingSum += Input(i);
  }
  void Initialise() {
    MovingSum = 0.0f;
    for(int i = -Input.Radius; i <= Input.Radius; ++ i)
      MovingSum += Input(i);
  }
  void Kernel() {
    MovingSum = MovingSum - Input(-Input.Radius-1) + Input(Input.Radius);
    Output() = MovingSum * MultBy;
  }
  const float MultBy;
  float MovingSum;
};
```
SIMT Optimisations – Block minimisation

- Thread block minimisation.
  - In simpler kernels, **thread scheduling overheads** can dominate.
  - One thread per pixel in a 4096x2304 image: **9.4M** threads.
  - A mapping of \( N \) output pixels to a single thread can alleviate this overhead.
SIMT Optimisations – horizontal rolling filters

- Threads move horizontally through the data
SIMT Optimisations – horizontal rolling filters

- Threads move horizontally through the data
- Reads are vertical
  - Inefficient, non-contiguous.
SIMT Optimisations – horizontal rolling filters

- Threads move horizontally through the data
- Reads are vertical
  - Inefficient, non-contiguous.
- Solve by reading a block
  - Limited shared memory makes this inefficient for a large number of threads.
SIMT Optimisations - Transposition

• Alternatively we can:
  • Transpose the dataset.
  • Make parallelism horizontal again, so reads are efficient.

• Transposition is as easy as adding transpose nodes to the DAG.
  • A post-optimisation looks for adjacent pairs of transpositions and remove them.
  • Thanks to DAG metadata.
SIMT Optimisations – Split row/column

- **Split row/column parallelism.**
  - Algorithms which are serialised in one axis may not be parallel enough.
  - GPUs keep **1000s** of threads in flight – images are not usually that **wide** or **tall**.
  - Parallelism can be “created” by *initialising* a new serialised run part-way.
  - Then **one thread** per axis becomes **two, three, four**, … with a small overhead.

```cpp
class BoxBlur : public Functor<BoxBlur, eMoving> {
    ...
    void Initialise() {
        MovingSum = 0.0f;
        for(int i = -Input.Radius; i <= Input.Radius; ++ i)
            MovingSum += Input(i);
    }
    ...
};
```
SIMT Optimisations – Access realignment

• Memory access realignment.
  • Additional requirement for **memory transaction grouping** in older hardware.
  • Thread 0, 16, 32, etc. must access a 16-element **aligned** region.
  • Images are appropriately aligned, but a **subregion** is probably not.
  • We can reassign the **thread:work mapping** to (mostly) fix this.

<table>
<thead>
<tr>
<th>N-3</th>
<th>N-2</th>
<th>N-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>..</th>
<th>..</th>
<th>..</th>
<th>..</th>
</tr>
</thead>
</table>

Realignment through Thread Renumbering
Performance Evaluation

- Realignment
- Thread Block Minimisation
- Split Column Parallelism
- Dual Transpose Elimination
- Staging
- Transposition
- No Optimisation

Diffusion Filtering

Degraining

x Speed-Up over Unoptimised

8800 GTX
GTX 260
Tesla C1060
8800 GTX
GTX 260
Tesla C1060
Performance Evaluation (Degraining)

Wavelet-Based Degraining

Throughput (MPix/s)

Image Size (MPix)

- Tesla C1060
- 8800 GTX
- Phenom 9650
- GTX 260
- Xeon E5420
- C2D E6600
Performance Evaluation (Degraining)

Out of Video Memory (Needs Host/GPU Tiling)

Wavelet-Based Degraining
Performance Evaluation (Diffusion Filtering)

![Graph showing throughput (MPix/s) vs. image size (MPix) for different hardware configurations including Tesla C1060, GTX 260, 8800 GTX, Xeon E5420, Phenom 9650, and C2D E6600.](image-url)
Performance Evaluation (Degraining)

GeForce 8800 GTX
Performance Evaluation (Diffusion Filtering)
Conclusions

• If performance requires parallelism and automatic optimisation:
  • Dependence information must be **built robustly into the code structure.**
  • Best effort parallelism cannot at present be relied upon.

• Metadata-supported frameworks **reduce** or **remove** the need for **code analysis**.
  • Trying to recover **high-level** algorithm concepts from an implementation is hard.
  • Many such concepts **embed** into the implementation **naturally**.
  • Source-to-source code generation allows **reuse** of **low-level** optimisations.

• Metadata are useful in a wide variety of optimisations.
  • In this presentation we outlined some optimisations for a **SIMT** architecture.
  • In previous work, we showed how metadata supports space and schedule optimisations to deliver large CPU speed-ups in visual effects DAGs.
  • In to-be-published work, we show how this framework supports **vectorisation** for SSE from the same source code (32 “cores”!).
Most importantly

- Metadata change the balance of development expertise.
  - High-performance software experts can work on the library framework.
  - Kernel authors can work on producing a kernel that generates the right result.
- Less developer time is used on platform-specific tuning.
- More time can be spent producing visual effects
  - Development effort targeted back to core values.
  - High performance still obtained.
- All of the work in this presentation is now moving from prototype to production.
  - Has been an opportunity to prototype our metadata and active libraries plans