VisionCPP: A SYCL-based Computer Vision Framework for Heterogeneous and Embedded Architecture

Mehdi Goli
Motivation

• Computer vision
  • Different areas
    • Medical imaging, film industry, industrial manufacturing, weather forecasting, etc.
  • Embedded systems
    • Automotive systems
    • Surveillance cameras
    • Challenge
      • Huge computational and communication demands
      • The stringent size, power and memory resource constraints
      • High efficiency and accuracy
  • Operations
    • Large size of data, Sequence of operations, Minimum operation time, Real-time operation
    • Potential suitable parallelism
      • Data & pipeline parallelism
Existing Frameworks

- OpenCV
  - Run-time optimisation
  - Adding custom function is hard
    - Eg. Channel level optimisation on GPU
  - Embedded systems
    - Not a trivial task

- OpenVX
  - Graph-based model
  - Limited number of built-in function
  - AMD has announced its implementation
  - No standard way of adding custom function
  - Every event has different way of adding custom function
Requirements

• Compile time optimisation
  • Specially embedded systems
• Easy to add custom function
• Unified front-end for different backend
• Cross-platform portability
• Performance Portability
SYCL

- Khronos group trademark
  - Royalty-free
  - Open standard

- Aim
  - Cross-platform abstraction layer
  - Portability and efficiency
    - OpenCL-enabled devices
  - “Single-source” style
  - Offline compilation Model

- Implementation
  - ComputeCPP (Codeplay)
  - TriSYCL (Open-source)
VisionCPP

• High-level framework
• Ease of use
  • Applications
  • Custom operations
• Performance portability
  • Separation of concern
  • No modification in application computation
    • OpenCL-enabled devices
    • CPU
VisionCPP Architecture

• **VisionCPP**
  • **Frontend**
    • Tree-based model
  • **Backend**
    • Compile-time optimisation
    • Offline C++ compilation
    • Predicable memory size
  • **Target**
    • Desktop
    • Embedded systems

Frontend Architecture
- Tree-based DSEL

Backend Architecture
- SYCL
- C++/OpenMP

Physical Hardware
- OpenCL-enabled device
- CPU
Tree Expression

- Colour Conversion Application:
  - Standard RGB $\rightarrow$ Linear RGB
  - Linear RGB $\rightarrow$ Linear HSV
  - Desaturation of $S$ Channel
  - Linear HSV $\rightarrow$ Linear RGB
  - Linear RGB $\rightarrow$ Standard RGB
**SYCL**

0: #include <visioncpp.hpp>
1: int main() {
2:  auto in = cv::imread("input.jpg");
3:  auto q = get_queue<gpu_selector>();
4:  auto a = Node<sRGB, 512, 512, Image>(in.data);
5:  auto b = Node<sRGB2IIRGB>(a);
6:  auto c = Node<IIRGB2IHSV>(b);
7:  auto d = Node<Constant>(0.1);
8:  auto e = Node<IHSV2Scale>(c, d);
9:  auto f = Node<IHSV2IIRGB>(e);
10: auto g = Node<sRGB2IIRGB>(f);
11: auto h = execute<fuse>(g, q);
12: auto ptr = h.get_data();
13: auto output = cv::Mat(512, 512, CV_8UC3, ptr.get());
14: cv::imshow("Display Image", output);
15: return 0;
16: }

**C++**

0: #include <visioncpp.hpp>
1: int main() {
2:  auto in = cv::imread("input.jpg");
3:  auto a = Node<sRGB, 512, 512, Image>(in.data);
4:  auto b = Node<sRGB2IIRGB>(a);
5:  auto c = Node<IIRGB2IHSV>(b);
6:  auto d = Node<Constant>(0.1);
7:  auto e = Node<IHSV2Scale>(c, d);
8:  auto f = Node<IHSV2IIRGB>(e);
9:  auto g = Node<sRGB2IIRGB>(f);
10: auto h = execute<fuse>(g, q);
11: auto ptr = h.get_data();
12: auto output = cv::Mat(512, 512, CV_8UC3, ptr.get());
13: cv::imshow("Display Image", output);
14: return 0;
15: }

[Diagram of Tree Expression]
VisionCPP frontend DSEL

- Tree-based Structure
  - Operation node
  - Leaf node
VisionCPP frontend DSEL

- Tree-based Structure
  - Operation node
    - Functors
      - operator()(T1 a);
      - operator()(T1 a, T2 b);
  - Leaf node

Node<Functor>(child_node);
VisionCPP frontend DSEL

• Tree-based Structure
  • Operation node
    • Functors
      • operator()(T1 a);
      • operator()(T1 a, T2 b);
  • Leaf node
    • SYCL Memory Model
      • Image (SYCL only)
      • Buffer (SYCL only)
      • Host (C++/OpenMP only)
    • Constant

Operation Node

Leaf Node

Node<Functore >(child_node);

Node<PixelType, Width, Height, MemoryModel >(ptr*);
Tree Expression(Functor)

0: `struct IHSV2Scale {`

1: `IHSV operator()( IHSV input, float coef) {`

2: `input.s() *= coef;`

3: `return input;`

4: `};`

5: `};`
Execution Policy: Fuse

- One Kernel
  - Apply the Functor operations
    - In serial order
    - Per pixel
Execution Policy: No Fuse

• Multiple Kernels
  • One pre non-terminal Node
  • Device only temporary output
    • on device global memory
Execution Policy: Custom Fuse

- Arbitrary Fusion
  - Any sub-tree
**Backend structure (For Fuse Policy)**

### SYCL

1: template<typename Expr, typename... Acc>

   void sycl (handler& cgh, Expr expr, Acc... acc) {

   // sycl accessor for accessing data on device
2:     auto outPtr = expr.out->template get_accessor<write>(cgh);  

   // sycl range representing valid range of accessing data
3:     auto rng = range < 2 > (Expr::Rows, Expr::Cols);

   // sycl parallel for for parallelising execution across the range
4:     cgh.parallel_for<Type>(rng), [=](item<2> itemID) {

6:       // calling the eval function for each pixel
7:       outPtr[itemID] = expr.eval (itemID, tuple);

8: });

### C++

1: template<typename Expr, typename... Acc>

   void cpp(Expr expr, Acc... acc) {

   // output pointer for accessing data on host
2:     auto outPtr = expr.out->get();

   // valid range for accessing data on host
3:     auto rng = range (Expr::Rows, Expr::Cols);

   // rebuilding the tuple of input pointer on host
4:     auto tuple = make_tuple (acc);

   // OpenMP directive for parallelising for loop
5:     #pragma omp parallel for
6:     for(size_t i=0; i < rng.rows; i++)
7:         for(size_t j=0; j < rng.cols; j++)

8:           // calling the eval function for each pixel
9:           outPtr[indx] = expr.eval (index (i, j), tuple);

};
Colour Conversion

- **Framework**
  - VisionCPP
    - Fuse
    - Custom Fuse
    - NoFuse

- **Platform**
  - Intel Core i7-4790K CPU 4.00GHz

---

**Execution Time (ms)**

<table>
<thead>
<tr>
<th>Image Size</th>
<th>VisionCPPFuse</th>
<th>VisionCPPCustomFuse</th>
<th>VisionCPPNoFuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.968</td>
<td>204.77</td>
<td>109.22</td>
</tr>
<tr>
<td>1024</td>
<td>2.096</td>
<td>109.22</td>
<td>57.291</td>
</tr>
<tr>
<td>2048</td>
<td>3.561</td>
<td>57.291</td>
<td>26.297</td>
</tr>
<tr>
<td>4096</td>
<td>7.811</td>
<td></td>
<td>49.254</td>
</tr>
</tbody>
</table>

---

codeplay.com
Colour Conversion

- Framework
  - OpenCV
  - VisionCPP
    - Fuse
- Platform
  - Intel Core i7-4790K
  - CPU 4.00GHz

![Bar Chart]

**Execution Time (ms)**

<table>
<thead>
<tr>
<th>Image Size</th>
<th>VisionCPPFuse</th>
<th>OpenCV</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.968</td>
<td>1.603</td>
</tr>
<tr>
<td>1024</td>
<td>3.579</td>
<td>4.513</td>
</tr>
<tr>
<td>2048</td>
<td>13.51</td>
<td>13.375</td>
</tr>
<tr>
<td>4096</td>
<td>57.291</td>
<td>56.402</td>
</tr>
</tbody>
</table>

*codeplay.com*
Colour Conversion

- Framework
  - OpenCV
  - VisionCPP
    - Fuse
- Platform
  - Oland PRO
    [Radeon R7 240]

![Execution Time Graph](image-url)
Colour Conversion

- **Framework**
  - VisionCPP
  - Fuse
  - OpenCV

- **Platform**
  - Oland PRO
    [Radeon R7 240]

- **Tool**
  - CodeXL

<table>
<thead>
<tr>
<th>Image Size for VisionCPP</th>
<th>Read Time (ms)</th>
<th>Write Time (ms)</th>
<th>RGB2HSV Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>0.1215</td>
<td>0.1253</td>
<td>0.1699</td>
</tr>
<tr>
<td>1024x1024</td>
<td>0.4813</td>
<td>0.4859</td>
<td>0.6387</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1.9135</td>
<td>1.9329</td>
<td>2.4655</td>
</tr>
<tr>
<td>4096x4096</td>
<td>8.3037</td>
<td>7.7967</td>
<td>10.3319</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Image Size for OpenCV</th>
<th>Read Time (ms)</th>
<th>Write Time (ms)</th>
<th>RGB2HSV Time (ms)</th>
<th>HSV2RGB Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>0.1246</td>
<td>0.1253</td>
<td>0.1338</td>
<td>0.1247</td>
</tr>
<tr>
<td>1024x1024</td>
<td>0.4744</td>
<td>0.4853</td>
<td>0.4905</td>
<td>0.4816</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1.8961</td>
<td>1.9286</td>
<td>2.0699</td>
<td>1.7486</td>
</tr>
<tr>
<td>4096x4096</td>
<td>7.6044</td>
<td>7.7913</td>
<td>8.2886</td>
<td>7.4403</td>
</tr>
</tbody>
</table>

Data Locality = \( \text{RGB2HSV} + \text{HSV2RGB} \) - \( \text{Data locality} \)
Conclusion

• The high-level algorithm
  • Applications
    • Easy to write
    • Domain-specific embedded language (DSEL)
  • Graph nodes
    • Easy to write
    • C++ functors

• The execution model is separated from algorithm
  • Portable between different programming models and architectures.
  • SYCL on top of OpenCL on heterogeneous devices

• The developer can control everything independently
  • Graphs, node implementations and execution model.

• Comparable Performance
Future work

• Histogram
• Optimise Neighbour operations
  • Nested Convolution
• Hierarchical parallelism
  • Pyramid
• Performance portability
  • Embedded system