codeplay

codeplay®

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

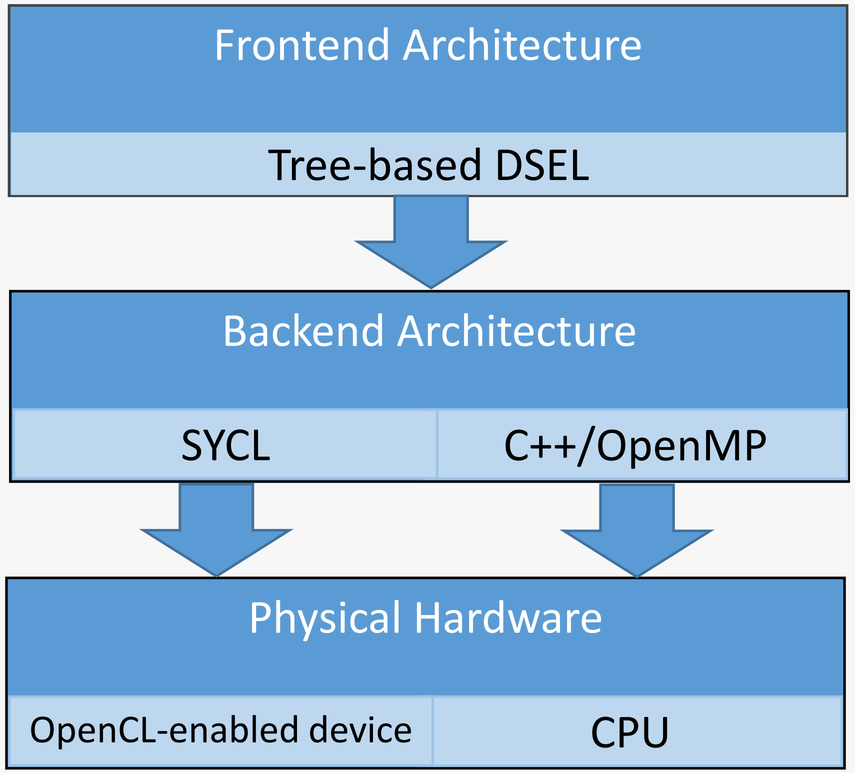
- Khronos group trademark
  - Royalty-free
  - Open standard
- Aim
  - Cross-platform abstraction layer
  - Portability and efficiency
    - OpenCL-enabled devices
  - “Single-source” style
  - Offline compilation Model
- Implementation
  - ComputeCPP (Codeplay)
  - TriSYCL (Open-source)

VisionCPP

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

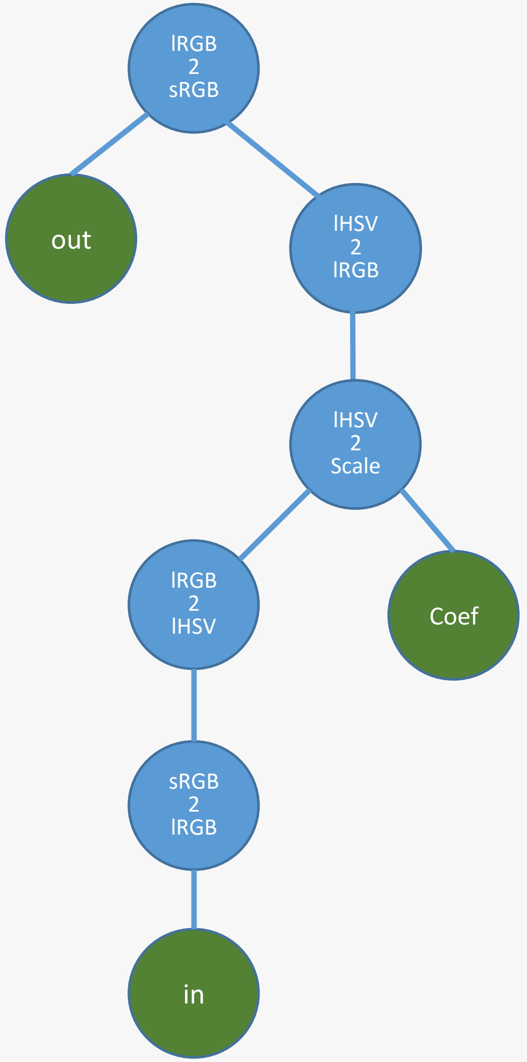
VisionCPP Architecture

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



Tree Expression

- Colour Conversion Application:
  - Standard RGB -> Linear RGB
  - Linear RGB -> Linear HSV
  - Desaturation of **S** Channel
  - Linear HSV -> Linear RGB
  - Linear RGB -> Standard RGB



Tree Expression

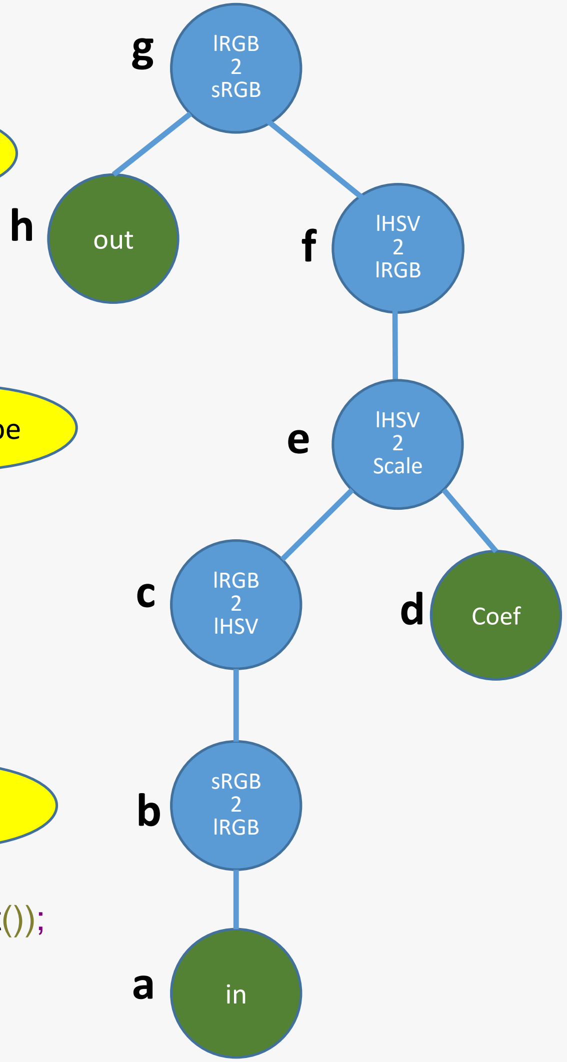
SYCL

```
#include <visioncpp.hpp>
int main() {
  auto in= cv::imread(“input.jpg”);
  auto q =get_queue<gpu_selector>();
  auto a = Node<sRGB, 512, 512,Image>(in.data));
  auto b = Node<sRGB2IRGB>(a);
  auto c = Node<IRGB2IHSV>(b);
  auto d = Node<Constant>(0.1);
  auto e = Node<IHSV2Scale>(c , d);
  auto f = Node<IHSV2IRGB>(e);
  auto g = Node<sRGB2IRGB>(f);
  auto h = execute<fuse> (g , q);
  auto ptr = h.get_data();
  auto output = cv::Mat(512 , 512 , CV_8UC3 , ptr.get());
  cv::imshow (“Display Image” , output);
  return 0;
}
```

Queue

Leaf Type

Queue



C++

```
#include <visioncpp.hpp>
int main() {
  auto in= cv::imread(“input.jpg”);

  auto a = Node<sRGB, 512, 512,Host>(in.data));
  auto b = Node<sRGB2IRGB>(a);
  auto c = Node<IRGB2IHSV>(b);
  auto d = Node<Constant>(0.1);
  auto e = Node<IHSV2Scale>(c , d);
  auto f = Node<IHSV2IRGB>(e);
  auto g = Node<sRGB2IRGB>(f);
  auto h = execute<fuse> (g );
  auto ptr = h.get_data();
  auto output = cv::Mat(512 , 512 , CV_8UC3 , ptr.get());
  cv::imshow (“Display Image” , output);
  return 0;
}
```

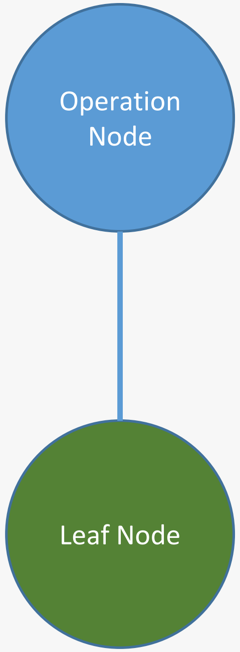
No Queue

Leaf Type

No Queue

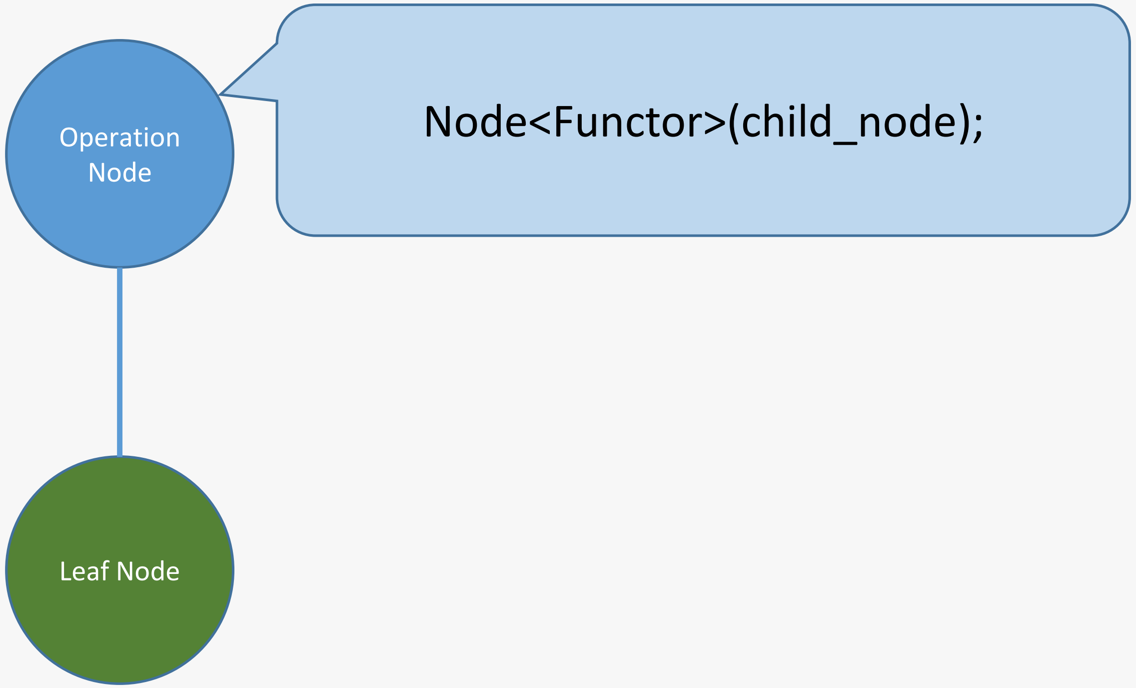
VisionCPP frontend DSEL

- Tree-based Structure
  - Operation node
  - Leaf node



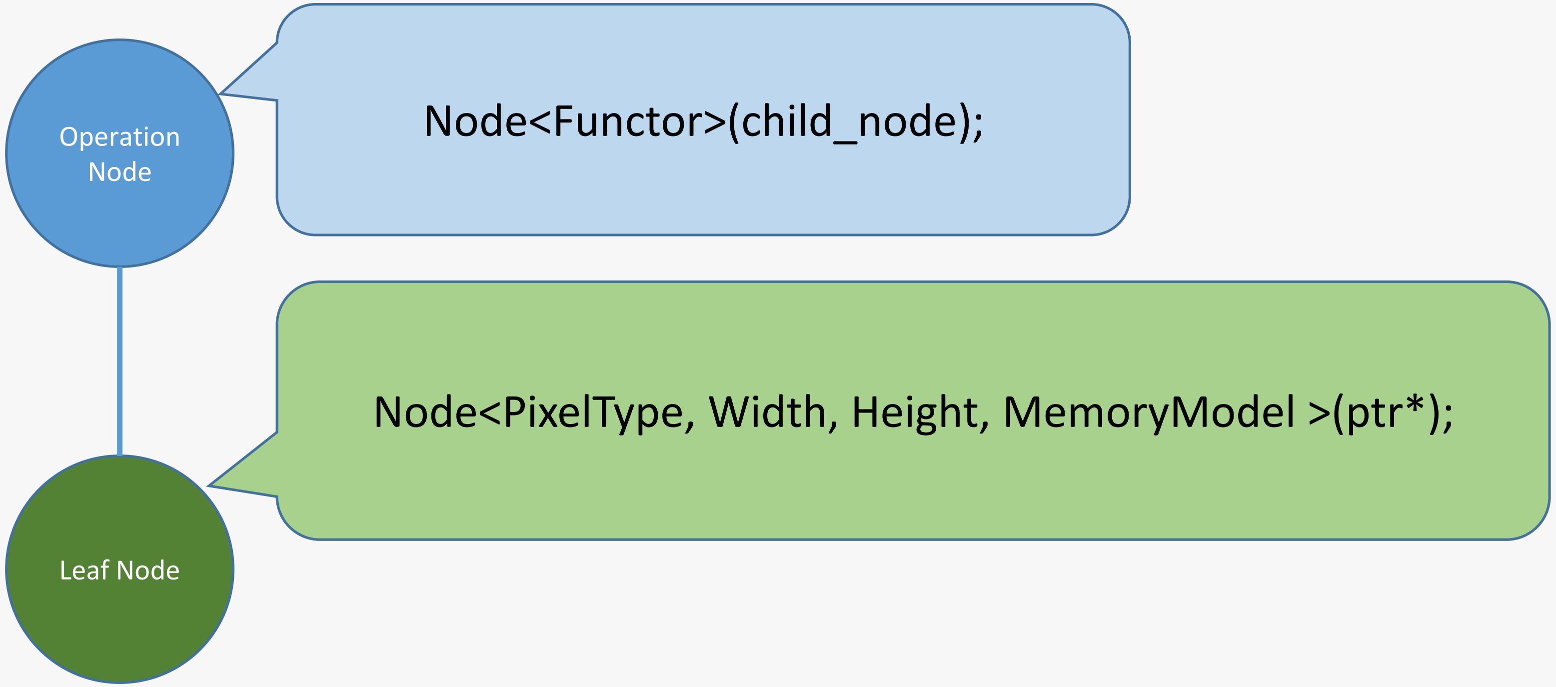
VisionCPP frontend DSEL

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

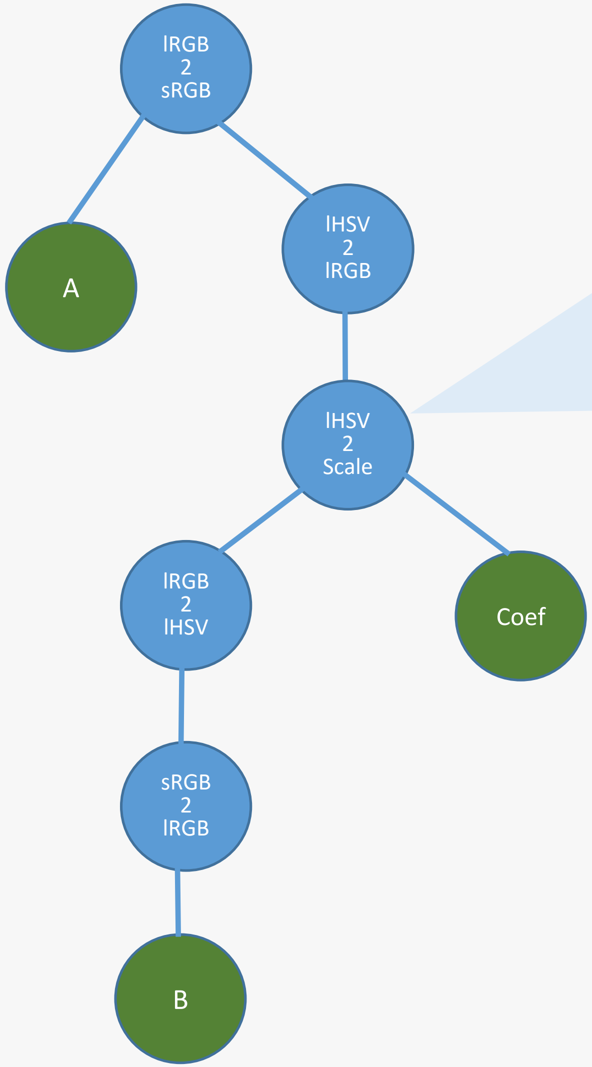


VisionCPP frontend DSEL

- 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



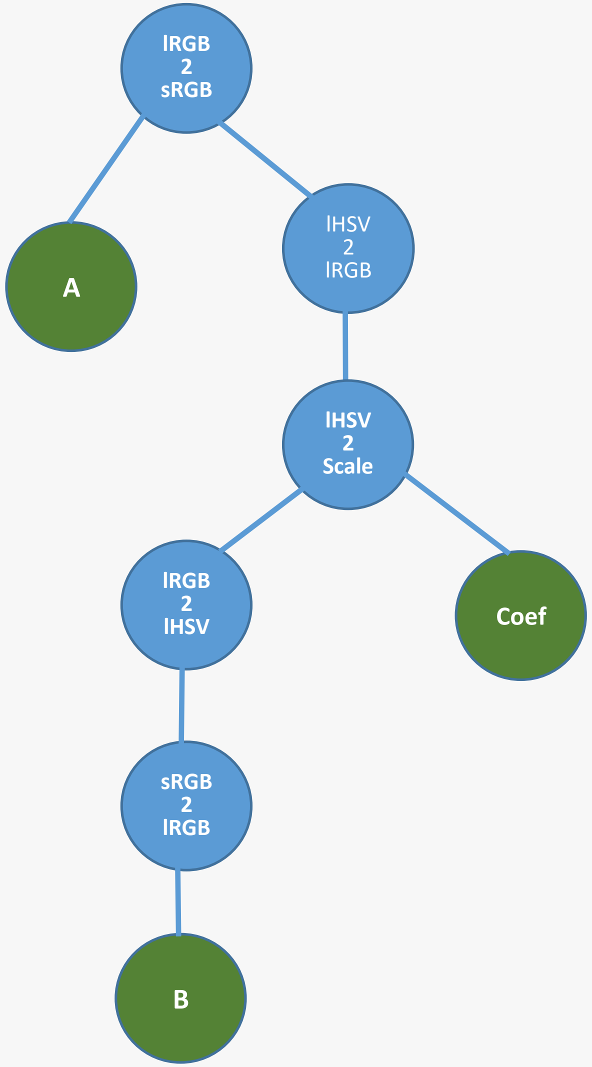
Tree Expression(Functor)




```
struct lHSV2Scale {
  lHSV operator()( lHSV input, float coef) {
      input.s() *= coef;
    return input;
  };
};
```

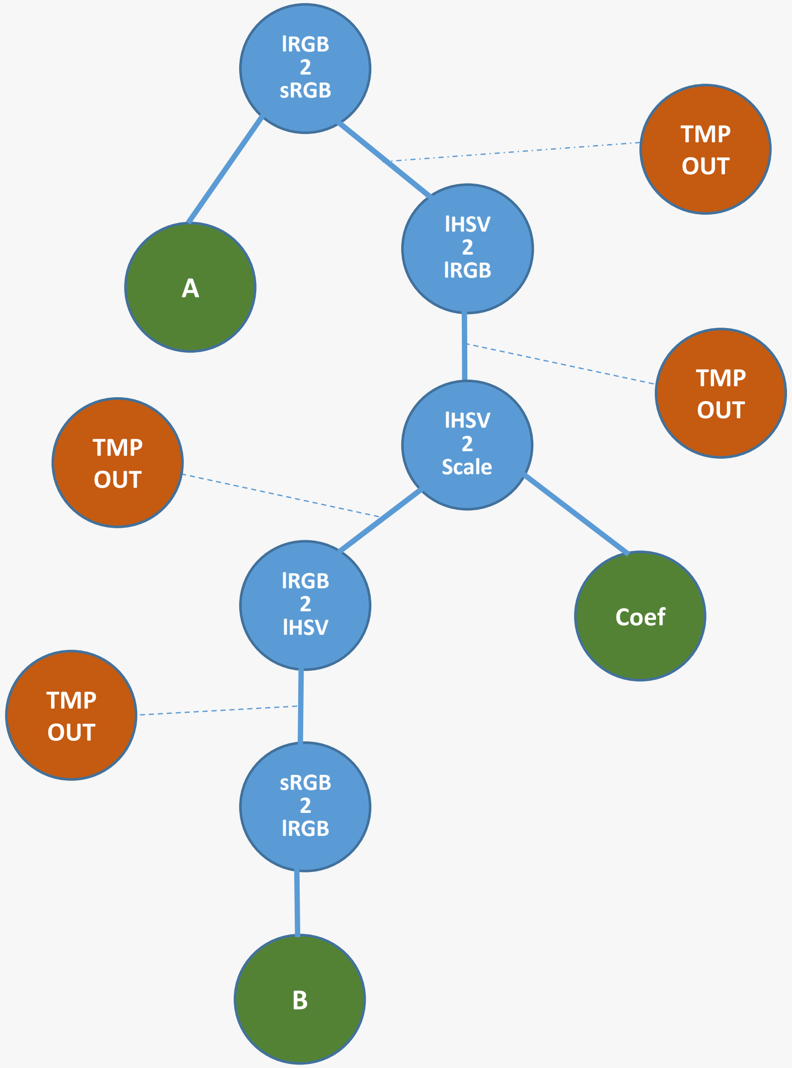
Execution Policy: Fuse

- One Kernel
  - Apply the Functor operations
    - In serial order
    - Per pixel



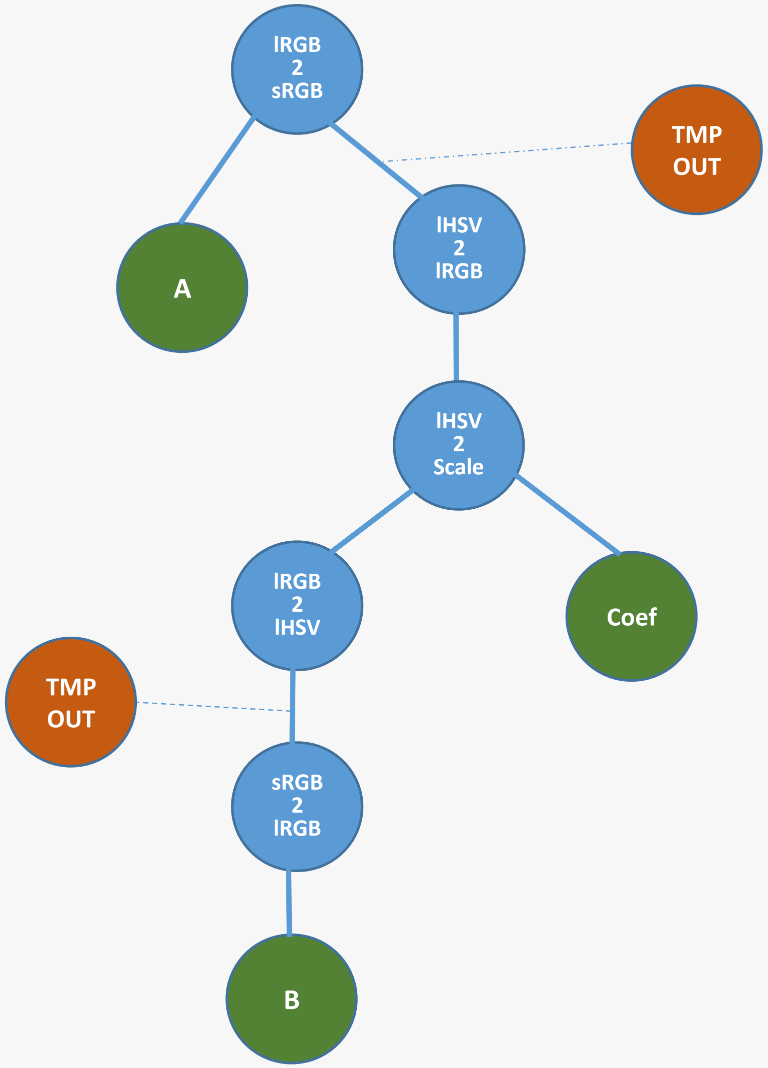
Execution Policy: No Fuse

- 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

```
template <typename Expr, typename… Acc>
void sycl (handler& cgh, Expr expr, Acc… acc) {

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

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

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

    // rebuilding accessor tuple on the device
    auto tuple = make_tuple (acc) ;

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

  });
}
```

Accessor

Parallel for

C++

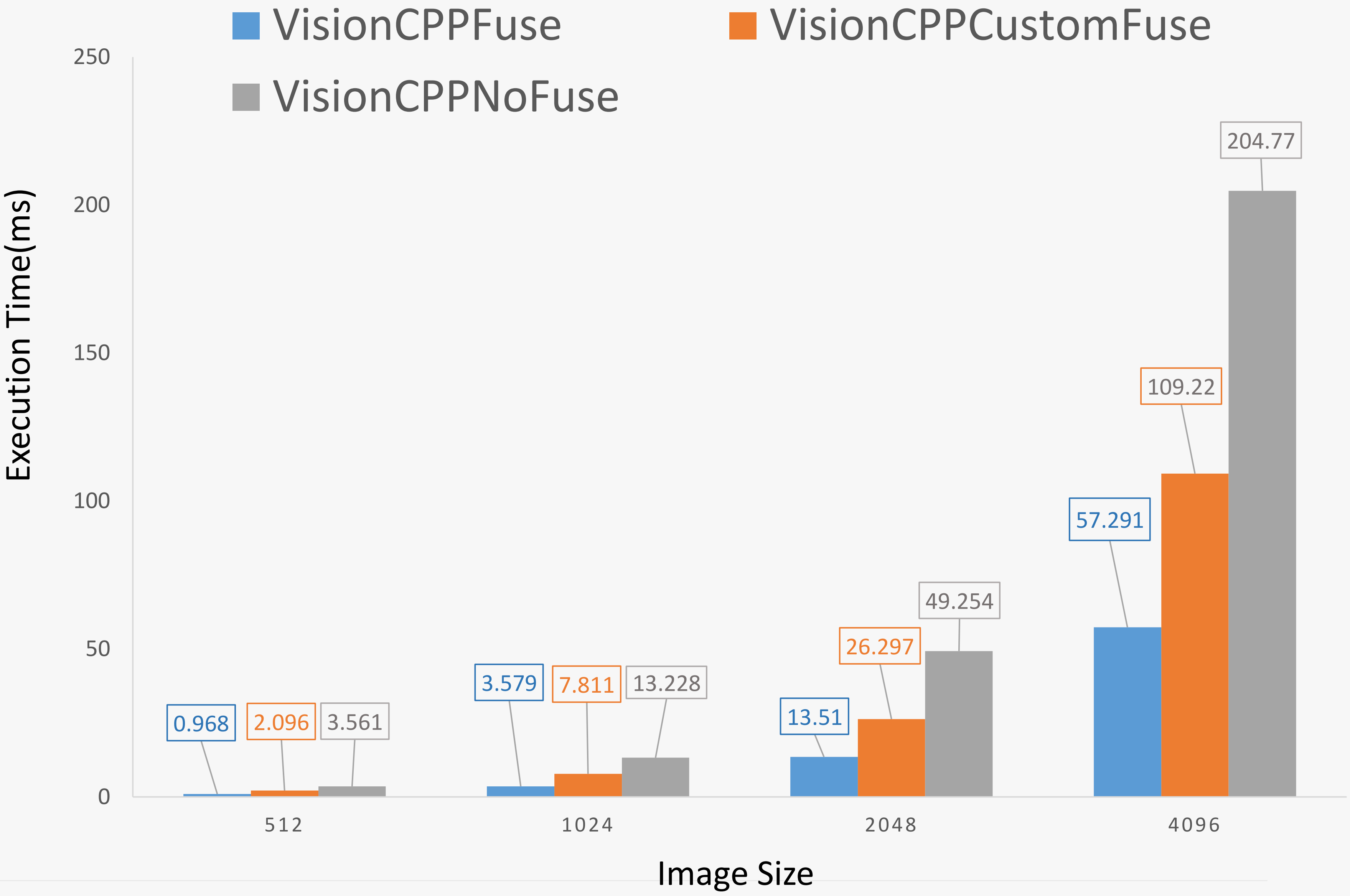
```
template <typename Expr, typename... Acc>
void cpp(Expr expr, Acc.. acc) {

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

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

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

  // OpenMP directive for parallelising for loop
  #pragma omp parallel for
   for(size_t  i=0;  i< rng.rows;  i++)
     for(size_t   j=0;   j< rng.cols;  j++)
       // calling the eval function for each pixel
       outPtr[indx] =  expr.eval (index (i , j), tuple );
};
```

Pointer

C++/OpenMP

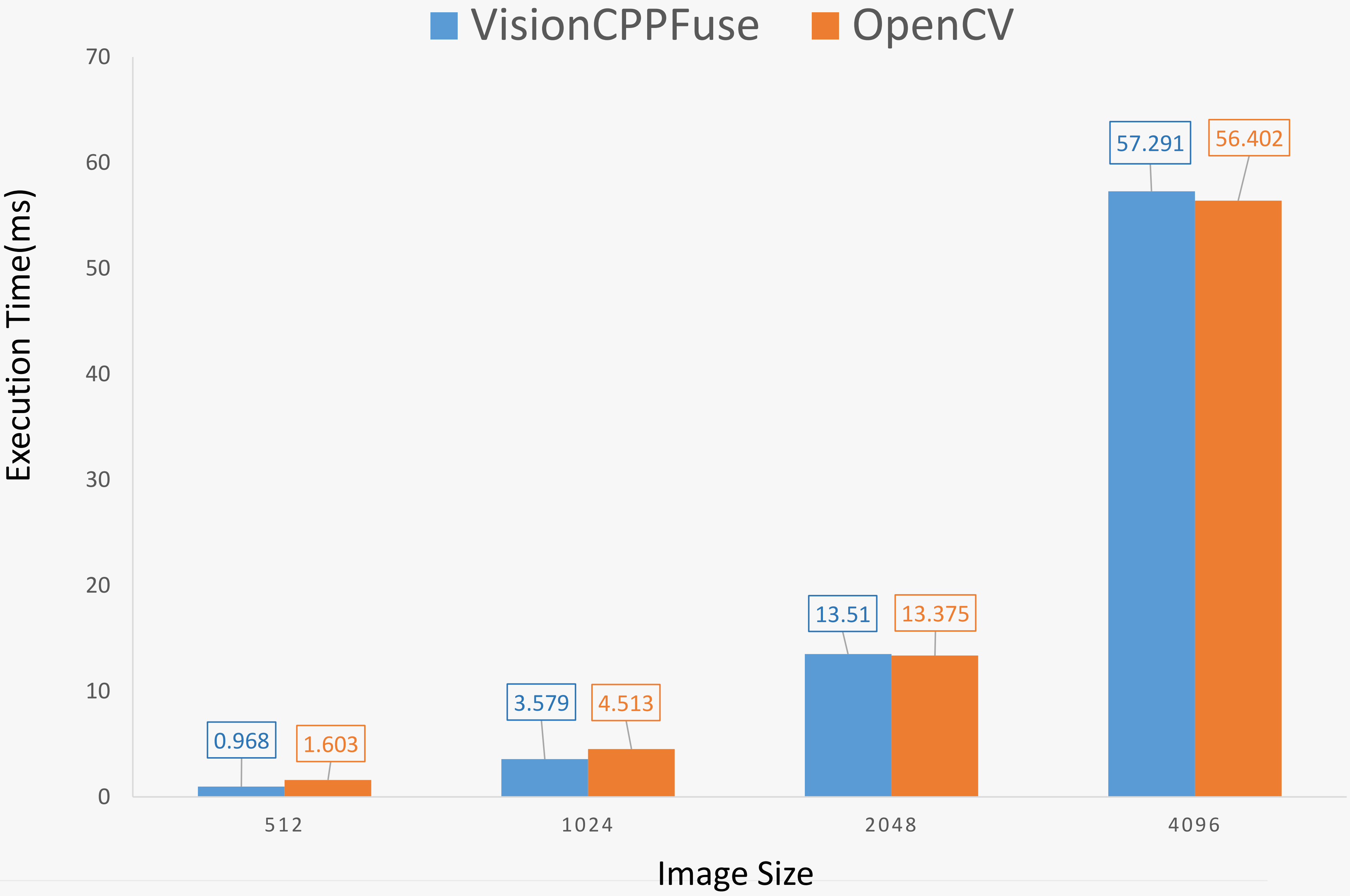
Colour Conversion

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



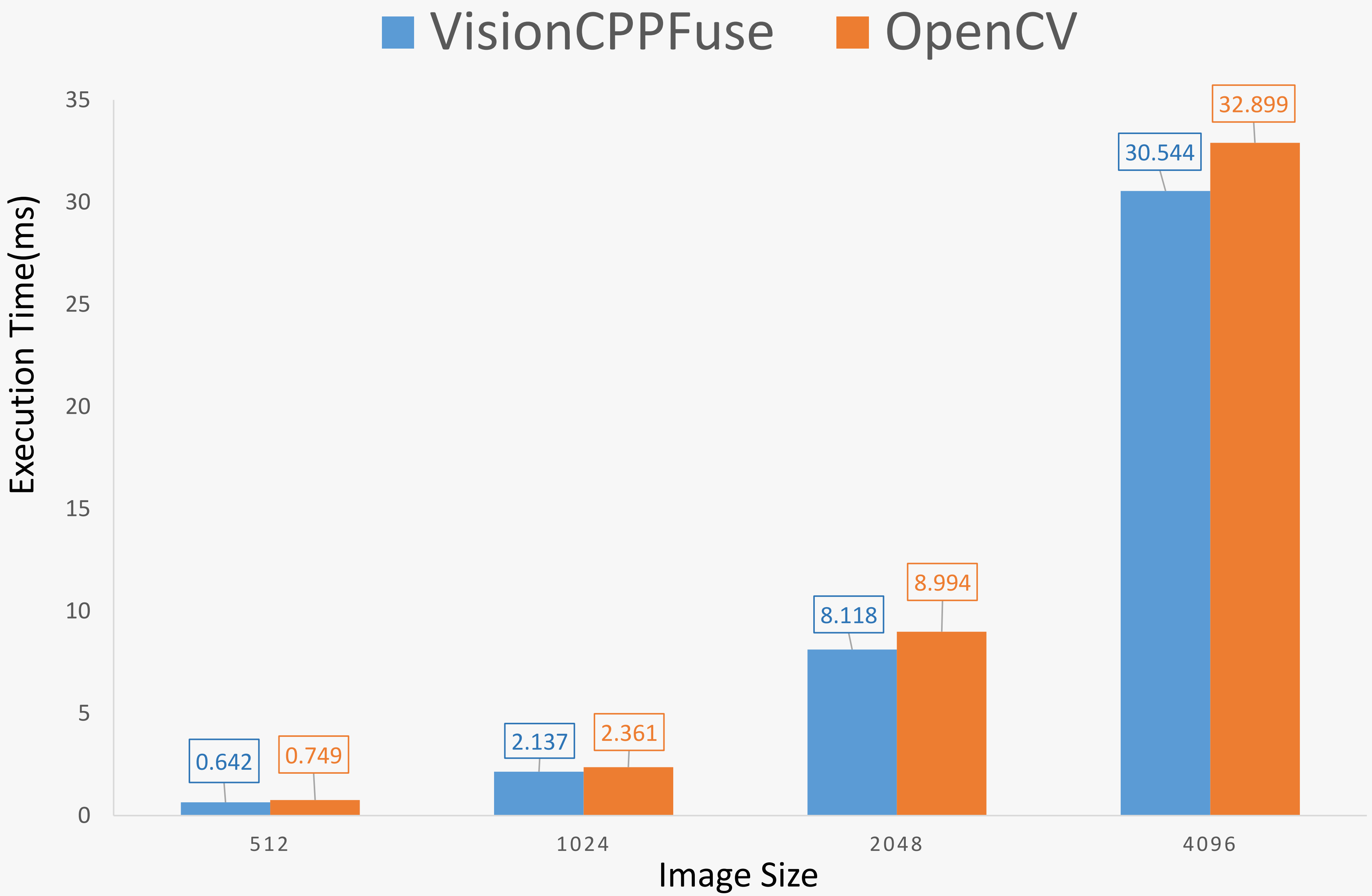
Colour Conversion

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



Colour Conversion

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



Colour Conversion

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

Image Size for VisionCPP	Read Time (ms)	Write Time (ms)	RGB2HSV HSV2RGB Time (ms)
512x512	0.1215	0.1253	0.1699
1024x1024	0.4813	0.4859	0.6387
2048x2048	1.9135	1.9329	2.4655
4096x4096	8.3037	7.7967	10.3319

DataLocality = RGB2HSVHSV2RGB - (RGB2HSV + HSV2RGB)	
Image Size for VisionCPP	Data Locality Time(ms)
512x512	0.0886
1024x1024	0.3334
2048x2048	1.353
4096x4096	5.397

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

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

We're Recruiting!

codeplay®

@codeplaysoft

info@codeplay.com