codeplay

VisionCPP

Mehdi Goli

Motivation

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

Motivation(continued)

- 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
    - Hard to get vendor implementation version
      - Sample version
    - No standard way of adding custom function
    - Every event has different way of adding custom function

Motivation(continued)

- SYCL
  - Khronos group
    - 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)

Vision Model

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

VisionCPP Model(continued)

- VisionCPP
    - Compile-time optimisation
        - System-level optimisation
            - Expression tree-based model
            - SYCL
        - Kernel-level optimisation
            - SYCL
            - OpenMP
        - Predictable execution time
            - No wait for compiling at run-time
                - SYCL
        - Predicable Memory size
    - Target
        - Desktop
        - Embedded systems

VisionCPP Model(continued)

- Tree-based Structure
  - Operation nodes
    - Vision library functors
  - Leaf nodes
    - Image
      - SYCL
    - Buffer
      - SYCL
    - Host
      - c/c++
      - OpenMP
    - Const
      - c/c++
      - SYCL
      - OpenMP

VisionCPP Example

```
//Including visioncpp Framework
#include <SYCL/ViLib.hpp>
int main() {
 //creating SYCL queue
cl::sycl::queue q;
 // creating leaf node from raw pointer
auto a= Node(visionMemory<512,
            512,TERMINAL::IMAGE, sRGB>(data));
 // creating constant variable
auto b= Node(visionMemory<TERMINAL::CONST>(0.1));
 // creating first operation node
auto c=Node<RGB2HSV>(a);
 //creating second operation Node
auto d=Node<HSV2SCALE>(c , b);
 // creating third operation node
auto e=Node<HSV2RGB>(d);
// executing the Pipeline
auto output = run(e , q);
 // getting the raw pointer on output
auto ptr=output.getData();
return 0;
}
```

SYCL Queue

Leaf Type

Execution Policy

```
//Including visioncpp Framework
#include <SYCL/ViLib.hpp>
int main() {

// creating leaf node from raw pointer
auto a= Node(visionMemory<512,
                         512,TERMINAL::HOST, IRGB>(data));
// creating constant variable
auto b= Node(visionMemory<TERMINAL::CONST>(0.1));
// creating first operation node
auto c=Node<RGB2HSV>(a);
//creating second operation node
auto d=Node<HSV2SCALE>(c , b);
// creating third operation node
auto e=Node<HSV2RGB>(d);
// executing the pipeline
auto output = run(e);
// getting the raw pointer on output
auto ptr=output.getData();
return 0;
}
```

Leaf Type

Execution Policy

IHSV2IRGB Functor

```
//Kernel struct and functor
#include <SYCL/ViLib.hpp>
struct IHSV2IRGB {
 IRGB operator()(IHSV input) {
   float fH, fS, fV fR, fG, fB;
   float fH = input.h; // H component
   float fS = input.s; // S component
   float fV = input.v; // V component
   float fI, fF, p, q, t; // Convert from HSV to RGB, using float ranges 0.0 to 1.0
    int iI;
   if( fS == 0 ) fR = fG = fB = fV; // achromatic (grey)
   else {
     If (fH >= 1.0f) fH = 0.0f; fH *= 6.0; // If Hue == 1.0, then wrap it around the circle to 0.0
     fI = floor( fH ); // sector 0 to 5
     iI = (int) fH;
     fF = fH - fI;
     p = fV * ( 1.0f - fS ); // factorial part of h (0 to 1)
     q = fV * ( 1.0f - fS * fF );
      t = fV * ( 1.0f - fS * ( 1.0f - fF ) );
      switch( iI ) {
        case   0:  fR = fV;   fG = t;    fB = p;    break;
        case   1:  fR = q;    fG = fV;   fB = p;    break;
        case   2:  fR = p;    fG = fV;   fB = t;    break;
        case   3:  fR = p;    fG = q;    fB = fV;   break;
        case   4:  fR = t;    fG = p;    fB = fV;   break;
        default:   fR = fV;   fG = p;    fB = q;    break;  } }
   return IRGB(bR,bG,bB); }; };
```

Backend Structure

```
template <size_t LeafType, typename Output,typename Expr,
                typename... rAccessors> void call_kernel(handler& cgh,Output&
                                outpt,Expr placeHolderExpr, rAccessors... rAcc) {
  constexpr size_t outTileSize = 16;
  constexpr size_t halo = 2;
  int inTileSize = outTileSize + (2 * halo);
  constexpr size_t xMode = (Output::Type::Rows) % outTileSize;
  constexpr size_t yMode = (Output::Type::Cols) % outTileSize;
  int xRange = Output::Type::Rows;
  int yRange = Output::Type::Cols;
  if (yMode != 0) yRange += (outTileSize - yMode);
  if (xMode != 0) xRange += (outTileSize - xMode);
  auto outPtr = (*(outpt.vilibMemory)).
                template getDeviceAccessor< access::mode::write>(cgh);
  cgh.parallel_for<typename TypeGenerator<Expr>::Type>(
                nd_range<2>(range<2>(xRange, yRange), range<2>(
                outTileSize, outTileSize)), [=](nd_item<2> itemID) {
  // Rebuild the tuple on the device
  auto device_read_tuple = make_tuple(rAcc...);
  // Eval, using compile time indices in the leaves to index
  ImageCoordinates imgCoordsGlobal(itemID.get_global(0),
  itemID.get_global(1));
  auto outval = placeHolderExpr.eval(imgCoordsGlobal,device_read_tuple);
  outPtr[itemID] =convert<typename decltype(outPtr)::value_type>(outval);
  });
}
```

SYCL Accessor

Parallel For

```
template <typename Output, typename Expr, typename... rParams>
void call_kernel(Output& outpt, Expr placeHolderExpr,
                                        Tuple<rParams...> device_read_tuple) {
 auto outPtr = (*(outpt.vilibMemory)).get();
 #ifdef _OPENMP
 #pragma omp parallel for default(none) shared(outPtr,\
                        device_read_tuple, placeHolderExpr)\
                        schedule(dynamic) num_threads(sysconf(\
                        _SC_NPROCESSORS_ONLN ))
 #endif
 for(size_t i=0; i< Output::Type::Rows; i++) {
  ImageCoordinates imgCoordsGlobal;
  imgCoordsGlobal.width=Output::Type::Rows;
  imgCoordsGlobal.height=Output::Type::Cols;
  for(size_t j=0; j< Output::Type::Cols; j++) {
   imgCoordsGlobal.x=i;
   imgCoordsGlobal.y=j;
   auto itemID=(i*Output::Type::Cols)+ j;
   auto outval = placeHolderExpr.eval( imgCoordsGlobal,

            device_read_tuple);
   outPtr[itemID] =convert<typename
            Dereference<decltype(outPtr)>::type>(outval);
  }
 }
};
```

Pointer

OpenMP

Case Study: GPU

- Framework
  - OpenCV
  - VisionCPP
- Platform
  - Oland PRO [Radeon R7 240]
- Image size:
  - 512x512

Kernel	OpenCV	VisionCPP(R)	VisionCPP(F)
lRGB2lHSV(ms)	0.1479	0.1336	…
lHSV2lRGB(ms)	0.1324	0.1213	…
Total(ms)	0.2803	0.2549	0.1898

Data Transfer	OpenCV	VisionCPP
Number of read	1	1
Number of write	1	1
Total read time(ms)	0.2401	0.2456
Total write time(ms)	0.2672	0.2779

Case Study: CPU

- Framework
  - OpenCV
  - VisionCPP
- Platform
  - Intel
    Core i7-4790K
    CPU 4.00GHz
- Compiler
  - Gcc-4.9.2
    - -O3
    - -mavx
    - Openmp 4.0 support
    - - fopenmp-simd
    - -mtune=intel
    - -march=native

Size	OpenCV-TBB(ms)	VisionCPP-SYCL Intel (F) (ms)	VisionCPP- SYCL Intel (R) (ms)	VisionCPP-OpenMP(F) (ms)	VisionCPP-OpenMP(R) (ms)
512x512	1.643	1.578	1.577	5.727	4.424
1024x1024	5.416	5.688	5.751	18.27	21.92
2048x2048	17.074	20.610	22.015	54.819	74.145
4096x4096	70.605	87.842	91.759	253.229	316.159
8192x8192	240.460	289.044	344.553	682.142	968.222

Conclusion

- The high-level algorithm
    - Applications
        - Easy to write
        - Domain-specific language (DSL)
    - 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
    - Pragma-based OpenMP.
- The developer can control everything independently
    - Graphs, node implementations and execution model.
- Comparable Performance

Future work

- Histogram
- Neighbour operation
  - Convolution
- Hierarchical parallelism
  - Pyramid
- Performance portability
  - Embedded system

We're Recruiting!

codeplay

Thanks for Listening!

@codeplaysoft

info@codeplay.com