Tracing and profiling dataflow applications

Pierre Zins, Michel Dagenais

December, 2017

Polytechnique Montréal

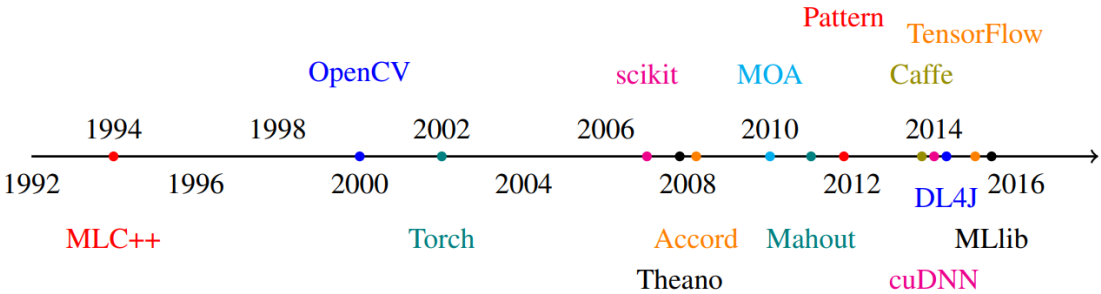
Laboratoire **DORSAL**

Agenda

- Introduction
- Existing tools for profiling
- Available platforms
- Current results
- Conclusion and future work

Introduction

Machine learning and deep learning



A tour of Tensorflow, Peter Goldsborough, Technische Universität München, oct 2016

- Extremely popular in many domains
- Quite expensive in terms of computation
- Always a need for better performance
- Should be able to train a model well on CPU + GPU platform

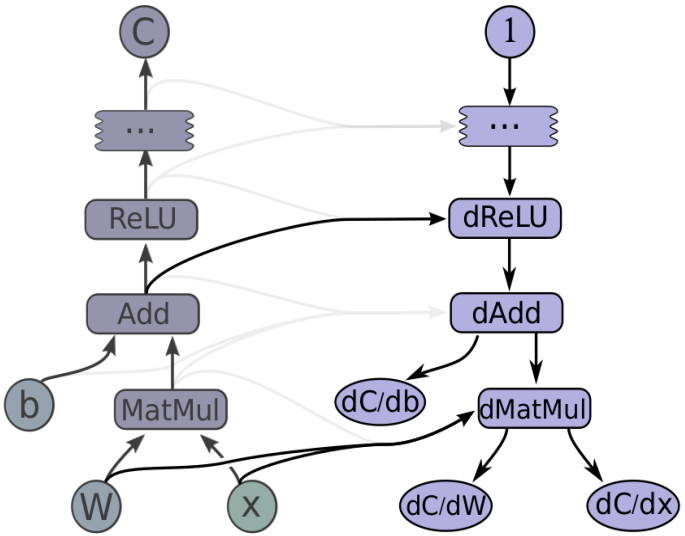
- Focus on Tensorflow
- Developed by Google
- Open-source since November 2015
- In October Tensorflow 1.4 has arrived

AMD R9 Nano
4096 Stream processors

- Training multilayer perceptron (50 epochs) :
  - CPU only : 240 s
  - CPU and GPU : 17 s

Introduction

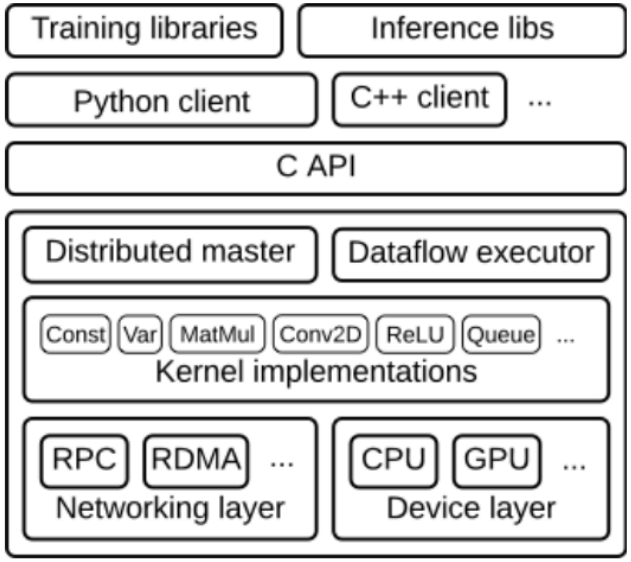
- Dataflow model
- Computation graph



Tensorflow : Large-Scale Machine Learning on Heterogeneous Distributed Systems, Abadi et al, November 2015

Research goals :

- Better understand the execution of the graph and its performance
- Insure that the GPU is used efficiently
- Detect problems or suggest optimizations



Tensorflow.org, architecture

Introduction

Tensorflow program : two phases

1) Definition of the computation graph

2) Execution of the graph with the provided input data

- Session : Links the python client and the core of Tensorflow

Tensorflow program pattern

Parameters : number of epochs, batch size, learning rate, ...

Network parameters : number of hidden layers, number of neurons, number of output classes, ...

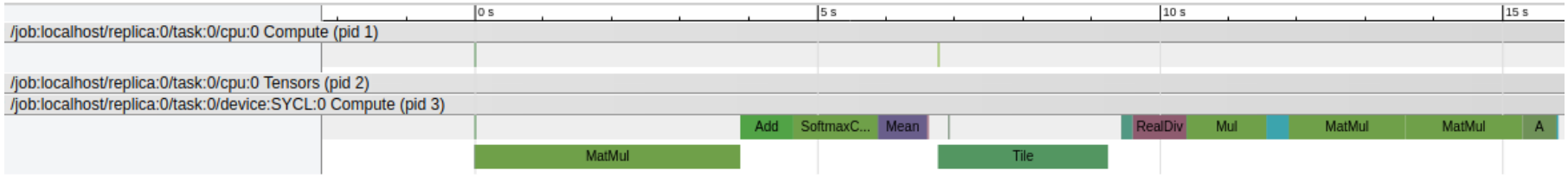
Graph definition : neurons, layers, activation functions, loss functions, optimizer, ...

```
with tf.Session() as sess:                                                    # create a sesion

    for epoch in range(training_epochs):                                      # loop number of epochs

        total_batch = int(mnist.train.num_examples/batch_size)
        for i in range(total_batch):                                          # loop on the dataset

            batch_x, batch_y = mnist.train.next_batch(batch_size)             # get batch data

            _, cost = sess.run([train_op, loss_op], feed_dict={X: batch_x, Y: batch_y}) # do the training
```

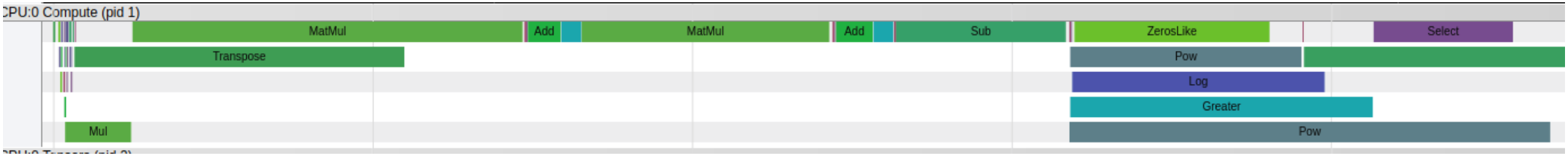
Existing tools for profiling

- Tensorflow internally collects metadata
- Limited to one session run
- Timeline module exports traces in Chrome Trace Format
- Visualization with the Chromium Viewer

```
...
{
    "ts": 1512158180650990,
    "pid": 1,
    "ph": "X",
    "cat": "Op",
    "tid": 0,
    "dur": 381,
    "name": "MatMul",
    "args": {
        "input0": "Add_1",
        "input1": "Variable_2/read",
        "name": "MatMul_2",
        "op": "MatMul"
    }
},
...
```

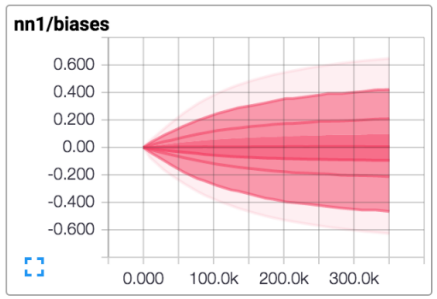


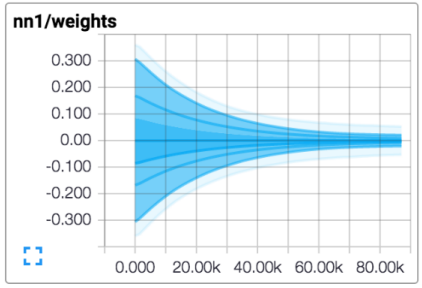
- CPU only



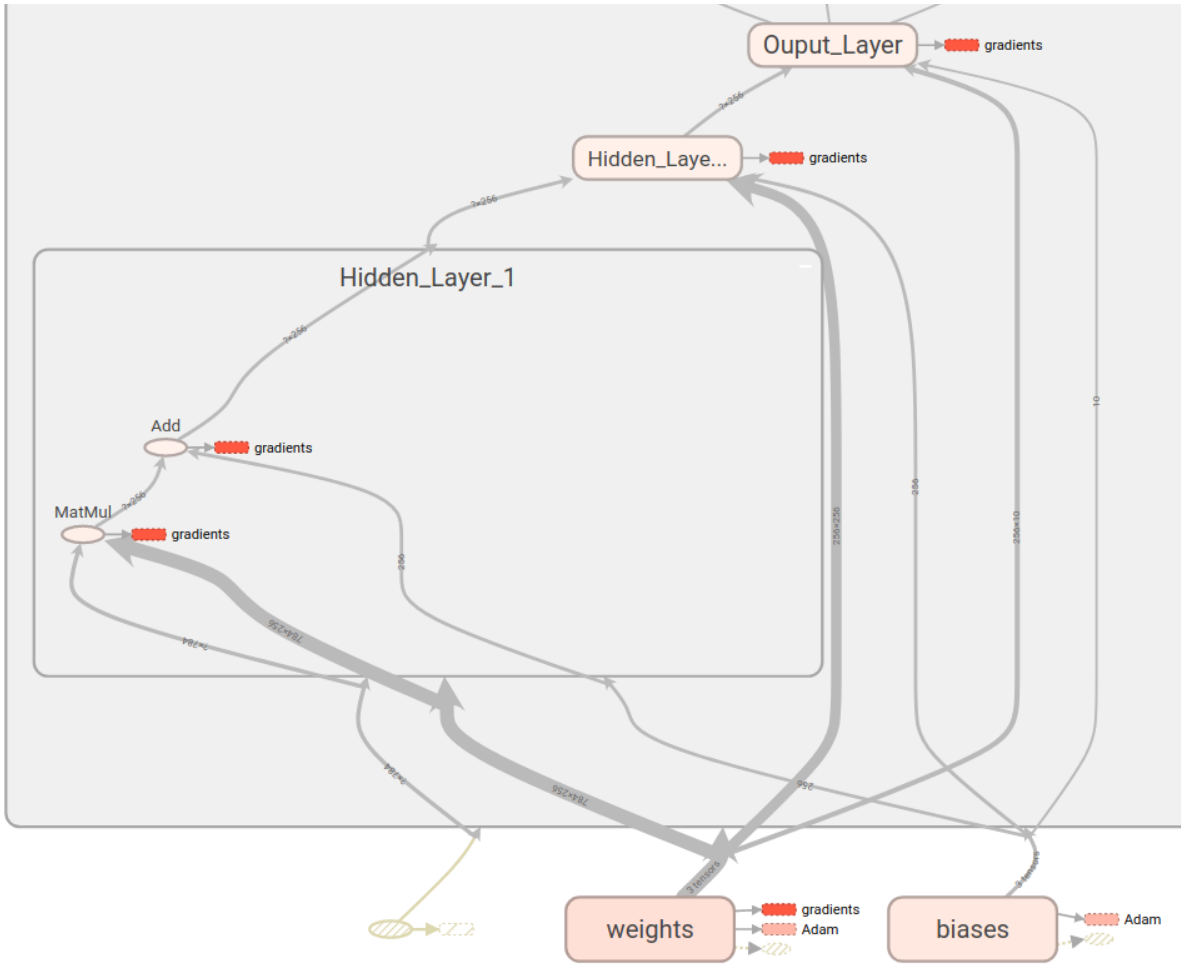
Existing tools for profiling

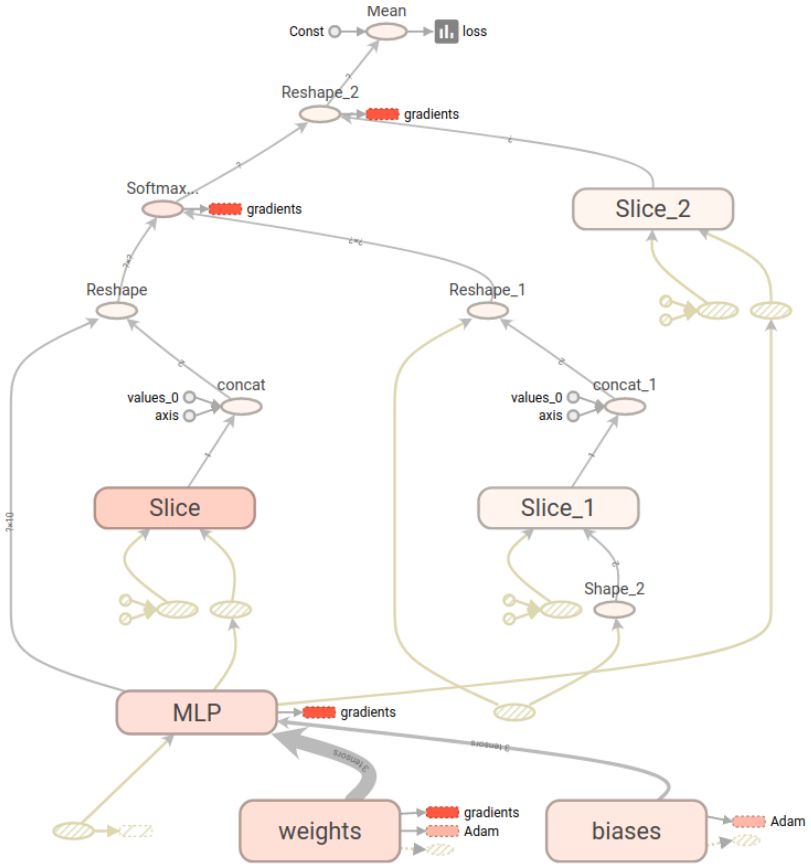
- Tensorboard : suite of visualizing tools
- Based on metadata collected during the execution





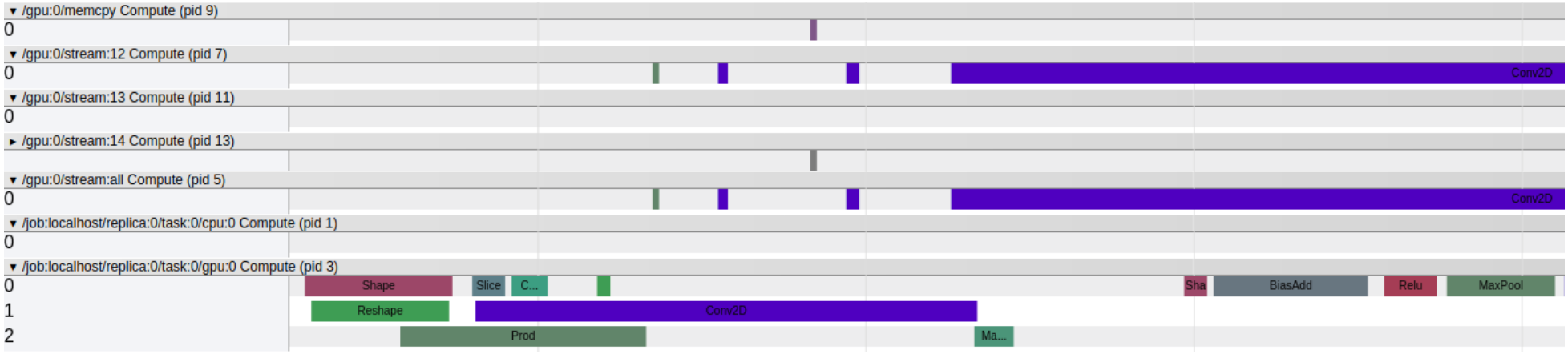
Tensorflow : Large-Scale Machine Learning on Heterogeneous Distributed Systems, Abadi et al, November 2015





GPU backend

- Initially : CUDA
  - Nvidia libraries : cuDNN, CUPTI
  - Requires Nvidia hardware
  - Additional information is collected



- Alternatives :
  - SYCL
  - HipTensorflow

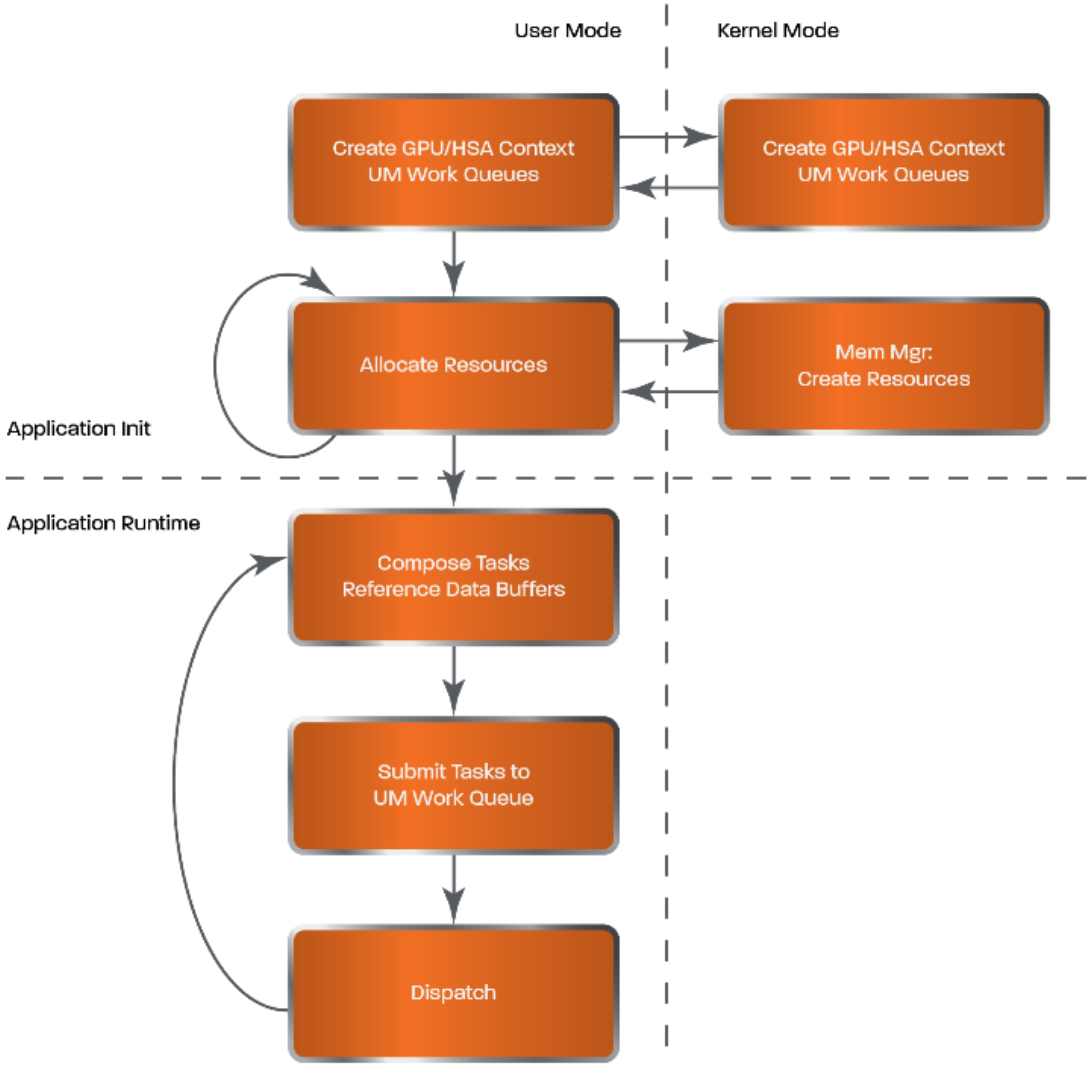
Alternatives - SYCL

- SYCL : C++ Single-source Heterogeneous Programming for OpenCL
- Tensorflow support of SYCL developed by Codeplay and Google
- Eigen support of SYCL
- Implementations : Codeplay LibComputeCpp, TriSYCL, SYCL-GTX
- Require proprietary OpenCL driver : AMD fglrx or AMDGPUPRO

```
template <typename T, size_t N>
void simple_vadd(const std::array<T, N> &A, const std::array<T, N> &B,
                 std::array<T, N> &C) {
  cl::sycl::queue deviceQueue;
  cl::sycl::range<1> numOfItems{N};
  cl::sycl::buffer<T, 1> bufferA(A.data(), numOfItems);
  cl::sycl::buffer<T, 1> bufferB(B.data(), numOfItems);
  cl::sycl::buffer<T, 1> bufferC(C.data(), numOfItems);

  deviceQueue.submit([&](cl::sycl::handler &cgh) {
    auto accessorA = bufferA.template get_access<sycl_read>(cgh);
    auto accessorB = bufferB.template get_access<sycl_read>(cgh);
    auto accessorC = bufferC.template get_access<sycl_write>(cgh);

    cgh.parallel_for<class SimpleVadd<T>>(numOfItems,
                [=](cl::sycl::id<1> wiID) {
            accessorC[wiID] += accessorA[wiID] + accessorB[wiID];
    });
  });
  deviceQueue.wait_and_throw();
}
```

Alternatives - HSA

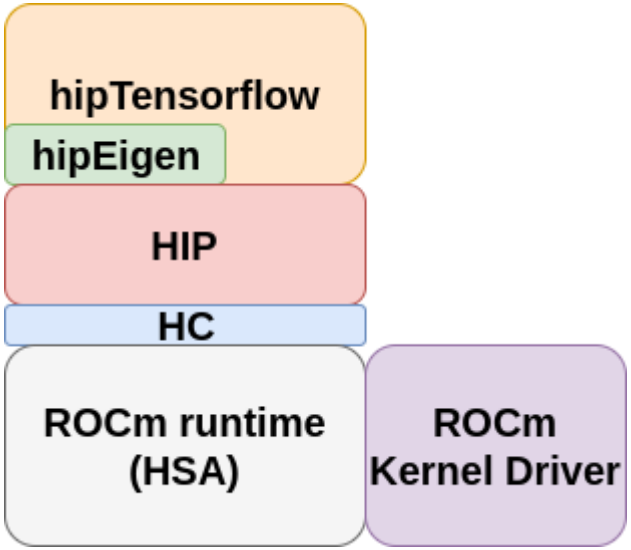
- HipTensorflow from AMD
- Work on HSA platform
- Three main advantages :
  - Single address space accessible to both CPU and GPU
  - User-space queues
  - Preemptive context switching



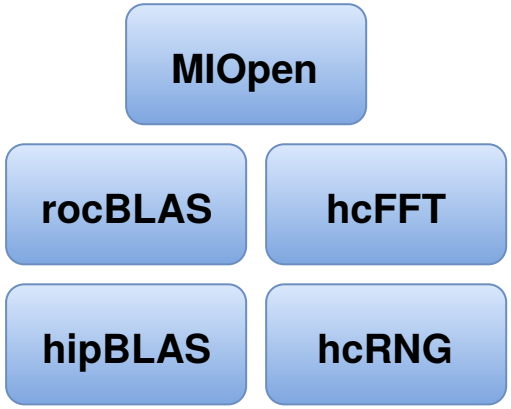
Heterogeneous System Architecture: A Technical Review, George Kyriazis, AMD, 2012

Alternatives - HSA

HipTensorflow



Other libraries from AMD



Compiler

Hipify

clang-based tool to automatically translate CUDA source code into HIP C++

Alternatives - HSA

- Profiling :
  - ROCm Profiler (RCP) : HSA API, kernel execution time, data transfers and performance counters
  - HIP Profiling : AMD markers for function entry and exit
  - HC Profiling : kernel execution, barrier, asynchronous copy

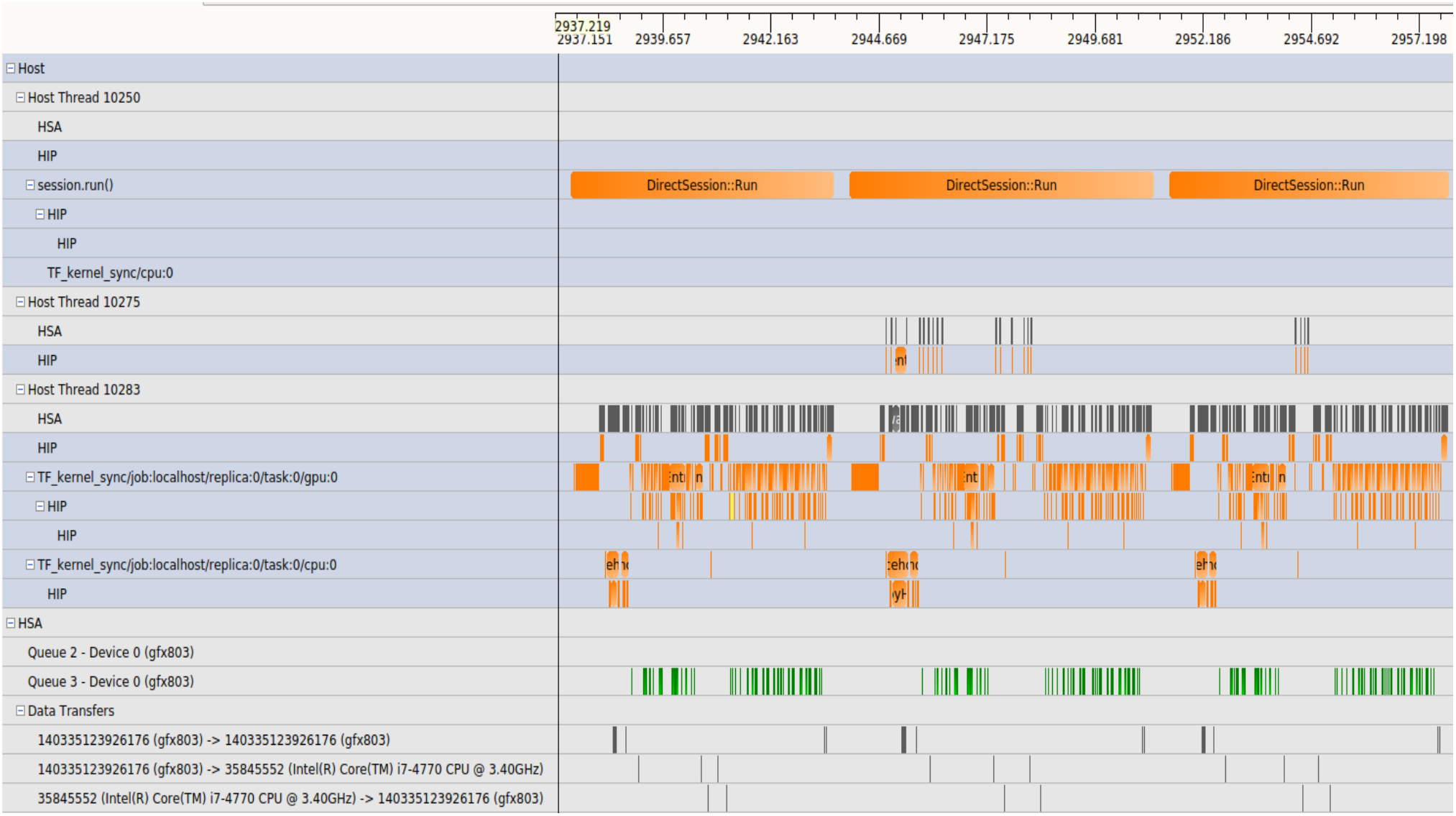
```
#define HIP_BEGIN_MARKER(markerName, group) amdtBeginMarker(markerName, group, nullptr);
#define HIP_END_MARKER(markerName, group) amdtEndMarkerEx(markerName, group, nullptr);
```

- Instrumentation of Tensorflow with markers

```
HIP_BEGIN_MARKER(op_kernel->name().c_str(), marker_name.c_str());
device->Compute(CHECK_NOTNULL(op_kernel), &ctx);
HIP_END_MARKER(op_kernel->name().c_str(), marker_name.c_str());
```

CODE XL

Alternatives - HSA



Alternatives - HSA

Performance counters

- ROCm-Profiler : flag -C
  - Get the counters for every kernel launched by the application
  - Some kernels require multiple passes
  - Not suitable for all applications

- GPUPerf API library (GPA)
  - Used by the profiler
  - More complex to use
  - More flexibility

tion	ThreadID	GlobalWorkSize	WorkGroupSize	VGPRs	SGPRs	Wavefronts	VALUInsts	SALUInsts	FetchInst	SFetchInsts	FetchSize	WriteSize	CacheHit (%)
1	5062	{ 512 1 1}	{ 256 1 1}	7	13	8	25.50	6.12	0	3.88	0.56	0.25	16.13
2	5086	{ 1024 1 1}	{ 512 1 1}	12	14	16	37	13	0	5	3.94	0	6.67
3	5086	{ 163840 1 1}	{ 512 1 1}	23	26	2560	805.63	64.87	0	10	15327.06	14611.62	1.83
4	5086	{ 163840 1 1}	{ 512 1 1}	12	14	2560	123.72	25.96	0	5	15316	14618	0.02
5	5086	{ 5120 1 1}	{ 1024 1 1}	6	11	80	37.55	8.96	0	4.97	0.81	0	4.40
6	5086	{ 30720 1 1}	{ 256 1 1}	19	20	480	5702.89	1162.30	45.40	6	15336.44	16.81	14.61
7	5086	{ 512 1 1}	{ 512 1 1}	21	25	8	10038.62	633.38	0	3.88	21.94	0	0
8	5086	{ 1024 1 1}	{ 512 1 1}	12	14	16	37	13	0	5	3.94	0	6.67
9	5086	{ 163840 1 1}	{ 512 1 1}	23	26	2560	805.63	64.87	0	10	15326.62	14606.62	1.88
10	5086	{ 163840 1 1}	{ 512 1 1}	12	14	2560	123.72	25.96	0	5	15316	14617.69	0.02

Example

autoencoder.py

- Training loop :

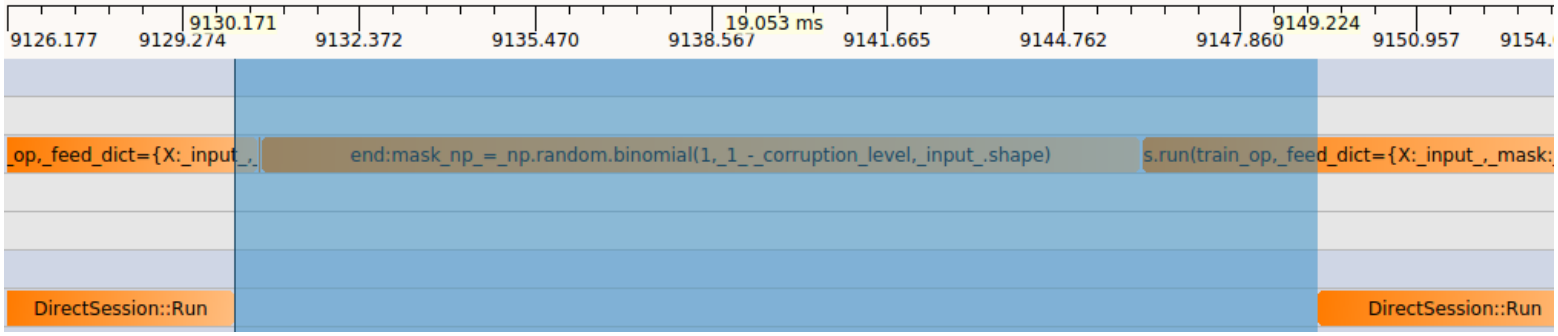
```
for i in range(10):
    for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(trX)+1, batch_size)):
        input_ = trX[start:end]
        mask_np = np.random.binomial(1, 1 - corruption_level, input_.shape)
        sess.run(train_op, feed_dict={X: input_, mask: mask_np})
```

Duration : 13.1 s

Can we improve it ?

Existing tools based on the collected metadata in Tensorflow are not useful in this case

Mean interval duration (530 session runs) : 19 ms



Example

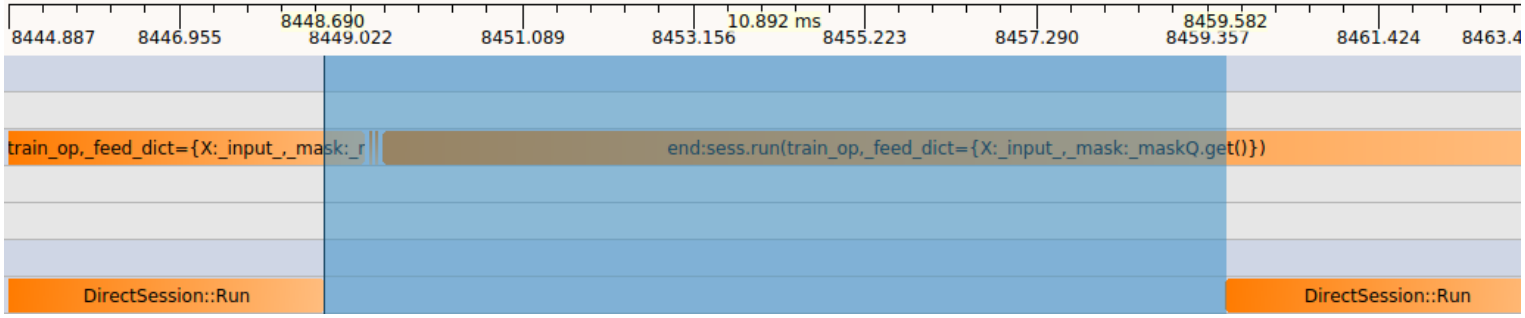
autoencoder.py

- Training loop :

```
maskQ = Queue()
def worker():
    while not is_end:
        mask_np = np.random.binomial(1, 1 - corruption_level, [1024, 784])
        maskQ.put(mask_np)

threads = []
for i in range(2):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

```
for i in range(10):
    for start, end in zip(range(0, len(trX), batch_size), range(batch_size, len(trX)+1, batch_size)):
        input_ = trX[start:end]
        sess.run(train_op, feed_dict={X: input_, mask: maskQ.get()})
```

Duration : 8.7 s

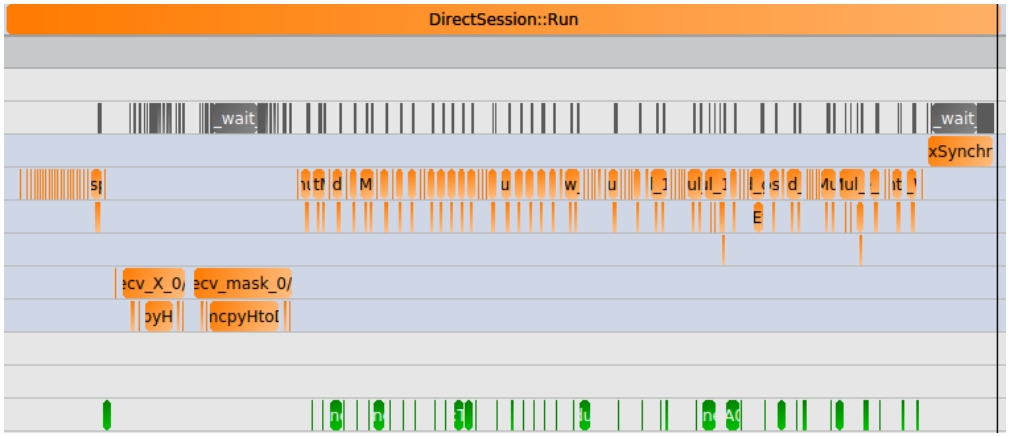


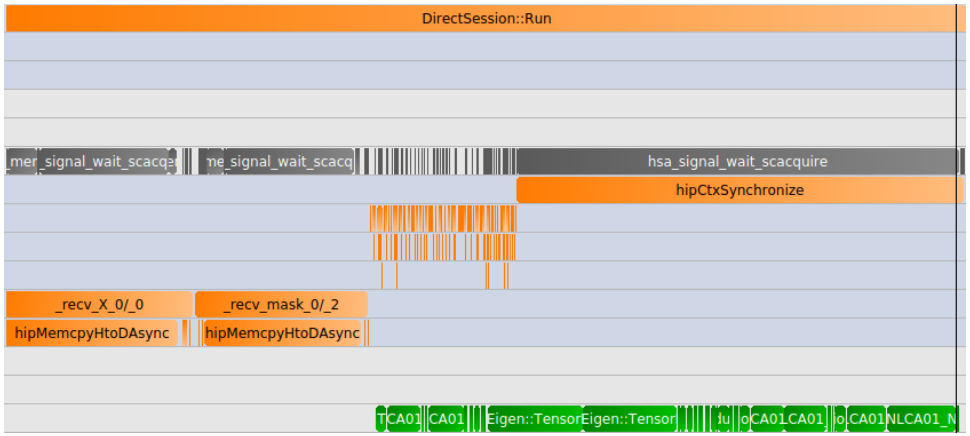
Mean interval duration (530 session runs) : 10.2 ms

Example

Autoencoder : GPU usage

- Change batch size to increase the GPU usage
- Limitation : can affect the learning, because bigger batches means less weights updates
- The difference is really obvious with the traces





Learn and stop when accuracy is higher than a threshold (0.9)

autoencoder

Batch size	Duration	GPU usage
32	144 s	18 %
64	104 s	21 %
80	93 s	23 %

multilayer perceptron

Batch size	Duration	gpu_usage
128	267 s	17 %
512	144 s	20 %
1024	75 s	22 %
2048	51 s	30 %

Example

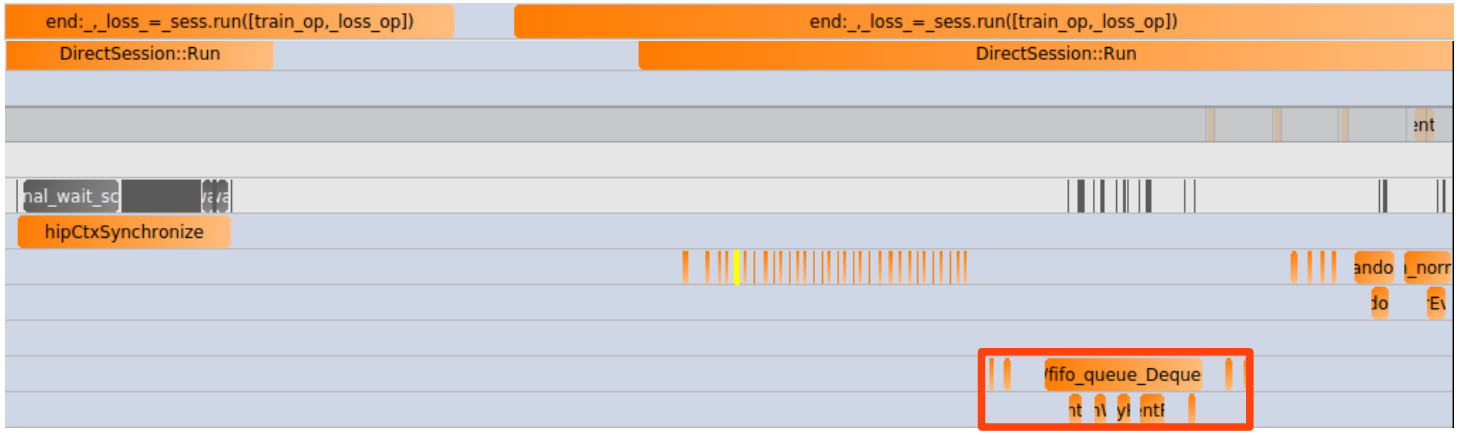
Input pipeline

- Existing tools are not suitable for analyzing the input pipeline
- The graph should not be waiting for new data

Using feed_dict

- Input data preparation : 1.5 ms
- Python feed_dict : 0.6 ms

- Execution time : 220s (50000 iterations)



Using Tensorflow Queue

- Dequeue data : 0.3 ms

- Execution time : 182 s (50000 iterations)

Conversion to CTF

- ROCm-profiler : outputs ATP format

- ATP : text format with 4 sections
  - API calls
  - API timestamps
  - Kernel execution timestamps
  - Performance markers

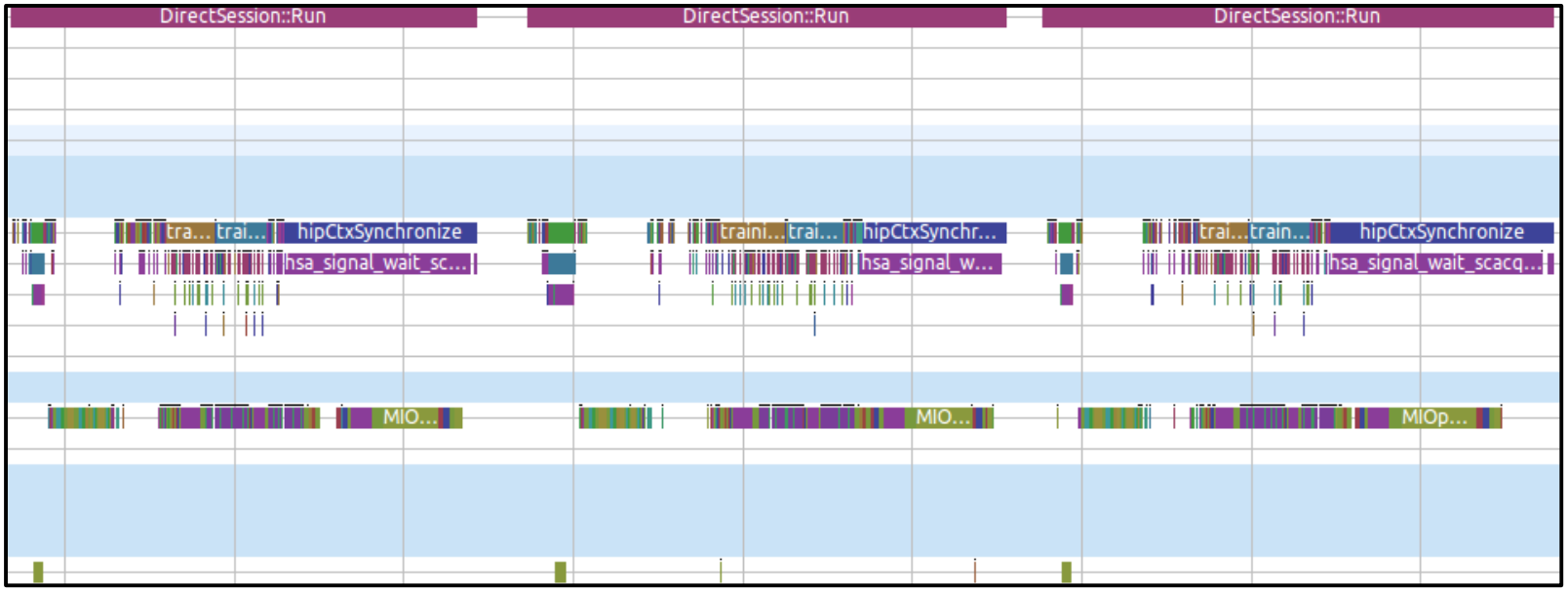
Elements	Method / instrumentation	Tools
Tensorflow	AMD markers	ROCm-Profiler
HIP	AMD markers	ROCm-Profiler
ROCm Runtime (HSA)	API Interception	ROCm-Profiler
Kernel execution	AMD special profiling functions	ROCm-Profiler
Data transfers	Deduced from HSA API calls	CodeXL
Python code	Lttng python logging	Lttng

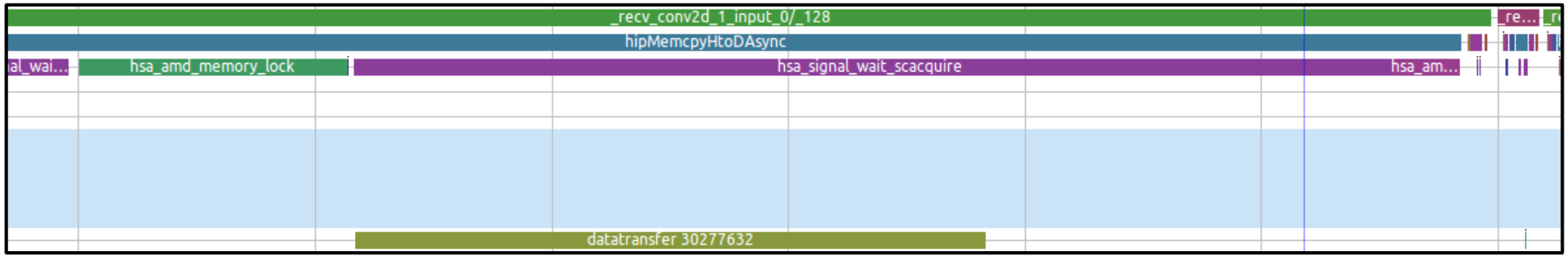
CodeXL:
- direct visualization of ATP traces
- difficulties with large traces (> 200-300 Mb)

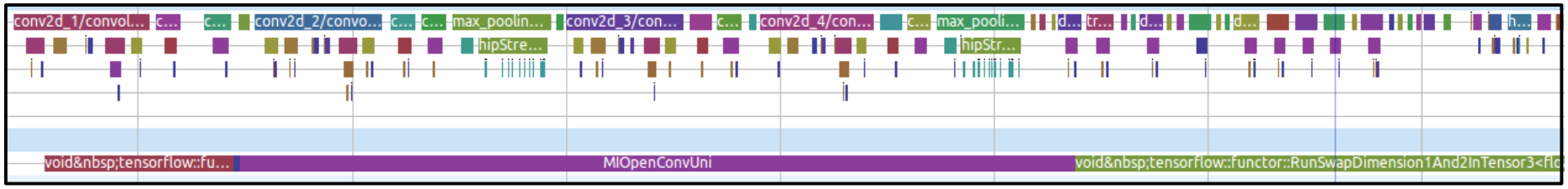
TraceCompass:
- manages well large traces
- combination with kernel traces

- Scripts to convert ATP into CTF using Babeltrace python bindings
- XML Callstack View for Tracecompass

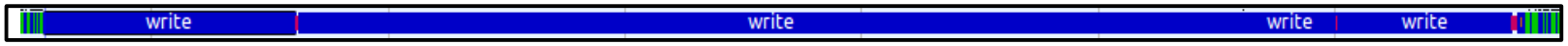
View TraceCompass







View TraceCompass



Conclusion

- Existing tools provide information for each node of the graph : execution time, memory usage
- Combining several profiling tools provide more detailed information
- Help to better understand the execution and the performance

Future work

- Investigate more the combination of kernel and userspace traces and try it on a non-HSA platform like CUDA or SYCL
- Integrate performance counters informations
- Compute more statistics from the trace
- Find more use-cases

Thank you!

Questions?