Computer Science Department

HPS

University of Reading

Exploiting heterogeneous resource utilisation for scientific workflows

Erdem G. YILMAZ, Julian M. KUNKEL

https://hps.vi4io.org

2020-04-23

LIMITLESS POTENTIAL | LIMITLESS OPPORTUNITIES | LIMITLESS IMPACT

Outline

University of Reading

Drivers of this research

University of Reading

- Data sizes are approaching exa-byte scale
  - Impossible to load/work on entire data set
- Its not feasible to move the data around
  - Network bandwidth becomes a bottleneck
- Varity of hardware resources increasing
  - Distributed Storage, Burst-buffers, NVRAM, SSD, Hybrid SSD+HDD, GPU
- Optimisation techniques are a domain on its own
  - Tuning HPC, edge-computing, cloud provisioning

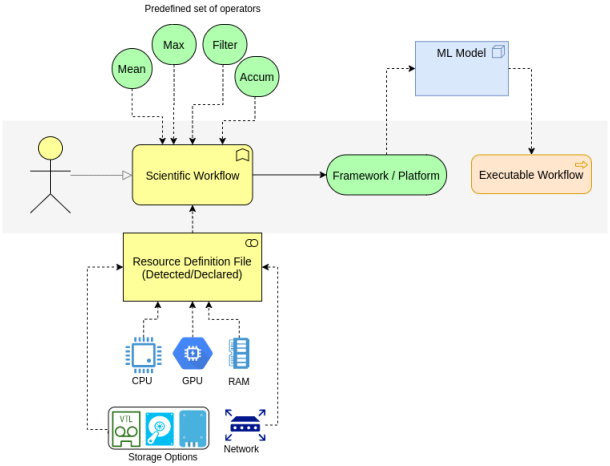
Expected outcome

University of Reading

- Code $\rightarrow$ Data solutions or data streaming
  - To avoid copying data around, code can be copied to node where data resides
  - Data, that doesnt fit into node memory, can be streamed to nodes.
- In-situ & In-transit processing techniques
  - Certain calculations/transformations will save compute/post-process cycle
  - Node local storage and computation to relieve network bandwidth
- Smart utilisation of heterogeneous hardware resources
  - Running code on CPU and/or GPU
  - Able to make a decision on which storage to be used
- Scientists should care less about optimisation and execution
  - Detect/Annotate underlying hardware resource capability
  - Running the workflow on best hardware mapping

Proposal

University of Reading



Declaration

- A set of operators will be created to be used in workflows, e.g. mean, max
  - Replicated for GPU and CPU
  - Able to publish provenance/performance data at runtime
  - Computational complexity and requirements of the operators are known
- User declares the workflow with its associated meta data
  - Operators and their inputs becomes tasks of a DAG workflow
  - Inputs identified, File type(NetCDF), location(local), size(100G)
  - Underlying hardware resources, declared or detected
    - Number of participating nodes and their specifications
    - Resources on each node, CPU, RAM, GPU, Net, Storage sepcs.

Proposal

University of Reading

Workflow Orchestration

- A machine learning model will be utilised to layout the actual workflow
  - Given the operators, inputs, hardware resources, where do we run the tasks
    - CPU and or GPU
    - Storage: node local, HDD, SDD, NVRAM, burst-buffer, distributed storage
    - Grouping of tasks to reduce communication
  - Execution metrics fed back to ML model for training
    - Network communication performance
    - Individual task timings
    - Storage performance metrics

Experiments

University of Reading

Experiments that have been conducted

- C++, python(single/multi-process), numpy, CUDA(cupy), Dask
- GPU streaming for comparison.
- GStreamer application, with two custom plugins
  - netcdf src plugin feeding lat x lon, 2D timeframes to accumulator
  - CUDA gpu accumulator for average operator over timeframes

Experiments planned for future

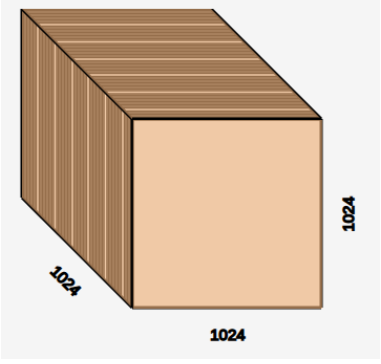
- Utilising DeepStream SDK from NVIDIA
- Streaming frameworks (Apache Flink or Apache Storm)

Setup

University of Reading

Input

- NetCDF file, 4.1G in size, stored on /dev/shm
- Single precision floats as temperature



The operation

- Mean of grid entries over all timeframes

The machine

- 132G free RAM
- 2x Intel Xeon(R) Silver 4108 CPU @ 1.80GHz, total 16 cores (32 with hpt)
- 1 GPU V100, 16G RAM, 5120 cuda cores

Python & Numpy

University of Reading

Single & multiprocess python

- Regular python looping as a single process, took **2804 sec.**

```
1    total =  np.zeros(shape=(lat_size,lon_size))
2    for lat_index in range(lat_size):
3        for lon_index in range(lon_size):
4            acc = 0
5            for i in range(tf_size):
6                acc += precip[i,lat_index,lon_index]
7            total[lat_index][lon_index] = acc
8    return total/tf_size
```

- Python looping with multiple processes, 30 processes, took **276 sec.**
  - Why not multi-threading? Global Interpreter Lock (GIL) won't let you

Numpy

- Same operation with numpy, took **10 sec.**
- Highly optimized C libraries under the hood.
- Has to load all data into memory, not feasible with larger data

```
1 from netCDF4 import Dataset
2 import numpy as np
3
4 ds = Dataset(filename)
5 precip = np.array(ds.variables["temp"][:])
6 precip.mean(axis=0)
```

cupy: numpy like operation with CUDA

- The mean() copy took **0.3sec.**
- Easy to work with, seemless gpu use, read+load+calc took **13 sec.**
- Requires all data to be loaded onto memory and then copied on to device
- Limiting factor is the file access, otherwise mean() is pretty fast.

```
1 from netCDF4 import Dataset
2 import numpy as np
3 import cupy as cp
4 ds = Dataset(filename)
5 with cp.cuda.Device(0):
6     temp = cp.array(ds.variables['temp'][:],copy=True)
7     ds.close()
8     temp_mean_device = temp.mean(axis=0)
9     cp.asnumpy(temp_mean_device)
```

C/C++

University of Reading

Single process

- Regular C/C++ code to perform the operation, took **11 sec.**

```
1         for (size_t rec_index = 0; rec_index < timeDimSize; rec_index++)
2         {
3             read_single_rec(precip, rec_index, chunk,....);
4             for (int lat = 0; lat < latDimSize; lat++) {
5                 for (int lon = 0; lon < lonDimSize; lon++) {
6                     accum[lat][lon] +=  chunk[lat][long];
7                 }
8             }
9         }
```

- NetCDF-C++ wrapper, regular triple for loop implementation
- What we loose in mean() operation, we gain in file reading

CUDA Kernel

- Same operation with numpy, took **11 sec.**
- Naive approach to GPU programming, heavily synchronized
- File I/O synched, hence not pleasingly parallel
- Under utilized CUDA cores.

CUDA Streaming

- Kernel execution and Host to Device memory copy, took **9 sec.**
- cudaDeviceSynchronize() had to be used for result integrity
- When executed with same file on NFS, it took **42 sec.**

CUDA Profiler

University of Reading

```
==30935== Profiling application: ./src/cuda_stream/cuda3 /dev/shm/test.nc temp
==30935== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   93.55%  5.14222s         4  1.28556s  1.28492s  1.28736s  mean(float*, int, int,
                    6.45%  354.28ms         5  70.856ms  356.19us  89.893ms  [CUDA memcpy HtoD]
                    0.01%  323.93us         1  323.93us  323.93us  323.93us  [CUDA memcpy DtoH]
      API calls:   80.80%  5.49503s         4  1.37376s  1.37279s  1.37617s  cudaDeviceSynchronize
                   13.01%  884.90ms         2  442.45ms  2.4915ms  882.41ms  cudaHostAlloc
                    6.02%  409.76ms         4  102.44ms  3.3230us  409.75ms  cudaStreamCreate
                    0.05%  3.6835ms         4  920.89us  2.7970us  3.3548ms  cudaFree
                    0.05%  3.5378ms         2  1.7689ms  269.38us  3.2685ms  cudaMalloc
                    0.02%  1.3426ms         4  335.64us  58.626us  1.0979ms  cudaLaunchKernel
                    0.01%  983.00us         2  491.50us  483.85us  499.15us  cudaMemcpy
                    0.01%  942.73us         1  942.73us  942.73us  942.73us  cuDeviceTotalMem
                    0.01%  358.99us        97  3.7000us     396ns  128.09us  cuDeviceGetAttribute
                    0.00%  338.79us         4  84.697us  55.239us  148.11us  cudaMemcpyAsync
                    0.00%  73.688us         4  18.422us  6.2320us  51.007us  cudaStreamDestroy
                    0.00%  35.923us         1  35.923us  35.923us  35.923us  cuDeviceGetName
                    0.00%  5.4540us         1  5.4540us  5.4540us  5.4540us  cuDeviceGetPCIBusId
                    0.00%  3.5760us         3  1.1920us     737ns  1.9240us  cuDeviceGetCount
                    0.00%  1.6760us         2     838ns     482ns  1.1940us  cuDeviceGet
                    0.00%     763ns         1     763ns     763ns     763ns  cuDeviceGetUuid
```

Improved kernel

- With better CUDA core utilisation and removed synchronisation, took **3 sec.**
- Host to Device copy took 350ms, kernel execution took 25ms
- Some further optimisation helped with overall timing
  - Major: Replacing triple loops with a flat array indexing
  - Minor: Use of pinned memory (page-locked memory)

```
1 for( int k=0; k < part; k++){
2     current_part = k*tcount;
3     float sum = 0;
4     for(int t=0; t < nrec; t++){
5         sum += x[tid + current_part + t * single_rec_size];
6     }
7     accum[current_part+tid] += sum;
8 }
```

CUDA Profiler Improved Kernel

University of Reading

```
==31220== Profiling application: ./src/cuda_stream_new_kernel/cuda5 /dev/shm/test.nc temp
==31220== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   93.30%  349.69ms        33  10.597ms  343.10us  11.066ms  [CUDA memcpy HtoD]
                    6.61%  24.786ms        32  774.55us  769.47us  782.62us  add(float*, int, int,
                    0.09%  342.27us         1  342.27us  342.27us  342.27us  [CUDA memcpy DtoH]
      API calls:   77.22%  371.79ms        32  11.618ms  2.8020us  370.64ms  cudaStreamCreate
                   18.50%  89.060ms         2  44.530ms  3.1113ms  85.949ms  cudaHostAlloc
                    2.59%  12.453ms         2  6.2265ms  420.23us  12.033ms  cudaMemcpy
                    0.50%  2.4198ms        32  75.618us  46.091us  387.21us  cudaMemcpyAsync
                    0.49%  2.3472ms        32  73.350us  45.408us  242.25us  cudaLaunchKernel
                    0.28%  1.3395ms         4  334.88us  2.1870us  1.0495ms  cudaFree
                    0.20%  945.70us         2  472.85us  257.69us  688.01us  cudaMalloc
                    0.11%  524.85us         1  524.85us  524.85us  524.85us  cuDeviceTotalMem
                    0.06%  294.41us        32  9.2000us  6.3500us  67.714us  cudaStreamDestroy
                    0.05%  254.75us        97  2.6260us     263ns  91.012us  cuDeviceGetAttribute
                    0.01%  25.416us         1  25.416us  25.416us  25.416us  cuDeviceGetName
                    0.00%  4.7820us         1  4.7820us  4.7820us  4.7820us  cuDeviceGetPCIBusId
                    0.00%  2.7620us         3     920ns     438ns  1.8240us  cuDeviceGetCount
                    0.00%  1.2090us         2     604ns     317ns     892ns  cuDeviceGet
                    0.00%     529ns         1     529ns     529ns     529ns  cuDeviceGetUuid
```

Dask + Xarray

- Took **12 sec.**
- Chunkwise operation enables us to work files larger than available mem size
- The chunks read by xarray. Local cluster of size 8 process, 32 threads
- Chunk sizes must be aligned with available memory on nodes.

```
from dask.distributed import Client
import xarray as xr
client = Client()
df = xr.open_dataset(filename,chunks={'time':128,'lat':128,'lon':128})
da = df['temp'].data
future = client.compute(da.mean(0)) # returns a future
future.result()
```

GStreamer

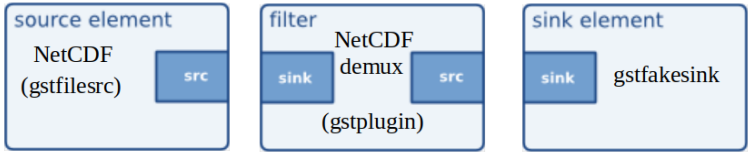
University of Reading

GStreamer as a data processing pipeline

- A framework to move data inside a pipeline from a source to a sink
- Each component implemens a well defined interface
- Generally used for encoded audio/video processing and display

GStreamer

University of Reading

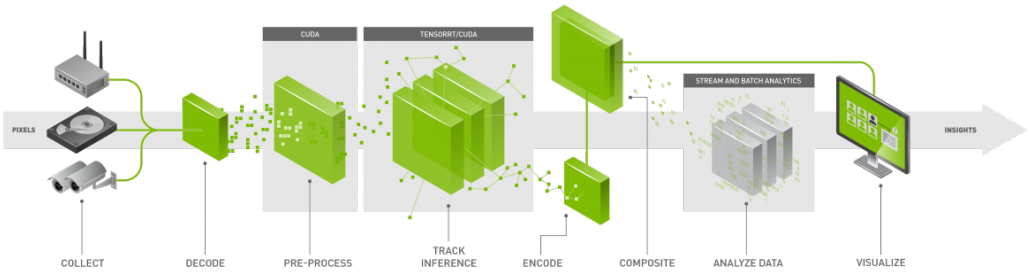


Two new plugins developed, netcdfsrc and netcdfdemux

- netcdfsrc: reads NetCDF file and sends to demux per timeframe
- netcdfdemux: accumulates incoming records and averages
- Ultimate aim was to interface NVIDIA DeepStream SDK

NVIDIA DeepStream

University of Reading



- A software SDK that utilizes CUDA through gstreamer interface
- Functionality revolves around video processing from multiple sources
- Pre-trained deep learning models, to detect features in a video stream

Next Steps

University of Reading

Mid term plans

- Complete the literature survey on workflows and resource mappings
- Interface NVIDIA DeepStream SDK with our NetCDF gstreamer plugins
  - Replacing video source with NetCDF file source plugin
  - Write a plugin that can process NetCDF content on CUDA hw
- Further tests with GPU and workflow languages (CWL, swift)
- Identify the shortcomings of the previous experiments

Long term plans

- Develop a prototype that is able to
  - Layout a workflow to resource mapping through ML model
  - Gather metrics and train the ML model

Summary

University of Reading

- Single proc python call, is horribly slow, took 2804s to complete
- Multiprocessor is relatively faster, took 276s with 30 process, 10x faster
- Numpy took 10 secs, 27x faster than the multiprocessor version
- cupy took 9 secs. as fast as numpy, Device-Host memcpy takes its toll
- C++ and Naive CUDA kernel are almost same with 11s
- Improved CUDA kernel is 3x faster with 3s.
- DASK + xarray, took 12 secs, able to cope with large files

Summary continued

University of Reading

- DASK performed slower compared to numpy but its distributed
- Numpy is limited with data sizes that can fit on the node's RAM
- CUDA is a faster alternative provided the problem is pleasingly parallel
- HPC/GPU tuning makes difference, its a complex domain of engineering