GVProf

Keren Zhou

Feb 21, 2023

GVPROF BASICS

GVProf is an advanced value profiler that locates value redundancy problems in GPU-accelerated applications. GVProf's code is available on Github.

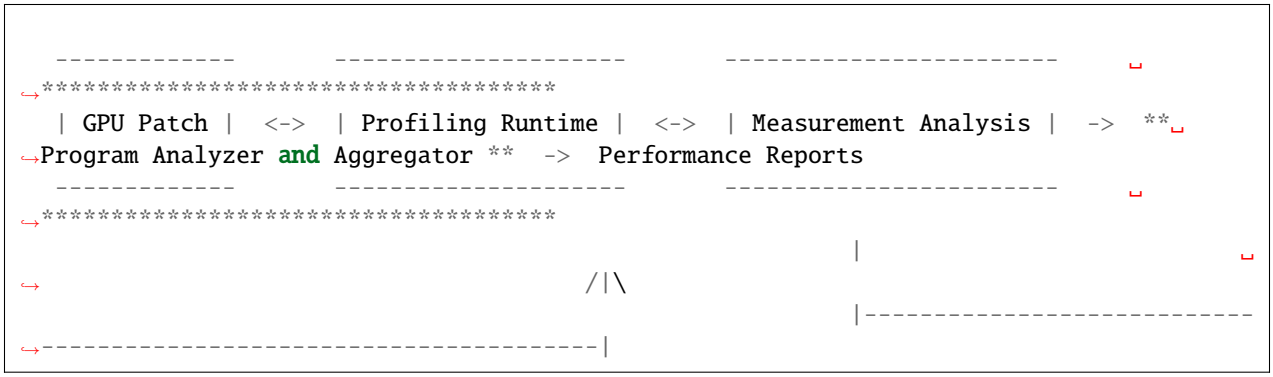
CHAPTER

ONE

PREFACE

GVProf is a memory profiling tool for applications running on GPU clusters, with advanced features for value based profiling and analysis.

The following diagram describes how components communicate with each other.



1.1 HPCToolkit (Profiling Runtime)

HPCToolkit is a powerful profiling tool that measures application performance on the world's largest supercomputers. GVProf customizes HPCToolkit and uses it as the default profiling runtime. Currently, we are developing on HPCToolkit's *sanitizer* branch.

1.2 Redshow

Redshow is a postmortem metrics analysis substrate. It receives data from the profiling runtime, performs analysis enabled by the user, and store the analysis result onto the disk. Besides, redshow maintains the information of data objects allocated at runtime. Redshow also contains binary analysis modules to map virtual addresses to function index and symbol names and analyze GPU instruction characteristics.

1.3 GPU Patch

GPU Patch includes several implementation of instrumentation callbacks and a GPU-CPU communication system. It can collect GPU memory metrics, block enter/exit records, and GPU call/ret records (under development). The collected data are stored on a GPU buffer. The profiling runtime observes a signal once the GPU buffer is full and copies data from the GPU to the CPU.

1.4 Program Analyzer and Aggregator

Some high level performance metrics are output to performance reports directly. Low level detailed performance metrics are associated with individual functions and lines. Therefore, we analyze program structure to attribute these metrics. Moreover, when analyzing application running on multiple nodes, we can aggregate performance data together to compute overall metrics that represent the entire execution.

CHAPTER

TWO

INSTALL

The documentation includes detailed instructions for every package required by gvprof. One can use `./bin/install` to install all these packages at once.

The install script accepts three arguments in order:

```
./bin/install <install-prefix> <path/to/cuda> <path/to/compute-sanitizer>
# default values
# <install-prefix>=`pwd`/gvprof
# <path/to/cuda>=/usr/local/cuda
# <path/to/compute-sanitizer>=<path/to/cuda>/compute-sanitizer
```

Before you install, make sure all the CUDA related paths (e.g., `LD_LIBRARY_PATH`) are setup.

2.1 GPU Patch

If you install cuda toolkit in somewhere else, you need to change the value of `SANITIZER_PATH`.

```
git clone --recursive git@github.com:Jokeren/GVProf.git
cd GVProf
make PREFIX=/path/to/gpu-patch/installation SANITIZER_PATH=/usr/local/cuda/compute-
↪sanitizer/ install
```

2.2 Dependencies

- spack

```
git clone https://github.com/spack/spack.git
export SPACK_ROOT=/path/to/spack
source ${SPACK_ROOT}/share/spack/setup-env.sh
```

- required packages

```
spack spec hpctoolkit
spack install --only dependencies hpctoolkit ^dyninst@master
# XXX(Keren): Temporary workaround until new hpctoolkit is merged
spack install libmonitor@master +dlopen +hpctoolkit
```

2.3 Redshow

```
cd redshow
# Tip: get boost libarary path 'spack find --path' and append include to that path
make install -j8 PREFIX=/path/to/redshow/installation BOOST_DIR=/path/to/boost/
↪installation GPU_PATH_DIR=/path/to/gpu-patch/installation
# Useful options:
# make DEBUG=1
# make OPENMP=1
```

2.4 HPCToolkit

- profiling substrates

```
cd /path/to/hpctoolkit
mkdir build && cd build
# Tip: check spack libraries' root->spack find --path.
# For example: --with-spack=/home/username/spack/opt/spack/linux-ubuntu18.04-zen/gcc-7.4.
↪0/
../configure --prefix=/path/to/hpctoolkit/installation --with-cuda=/usr/local/cuda-11.0 -
↪-with-sanitizer=/path/to/sanitizer --with-gpu-patch=/path/to/gpu-patch/installation --
↪with-redshow=/path/to/redshow/installation --with-spack=/path/to/spack/libraries/root -
↪-with
make install -j8
```

- hpcviewer (optional)

http://hpctoolkit.org/download/hpcviewer/

2.5 Setup and Test

Add following lines into your `.bashrc` file and source it.

```
export PATH=/path/to/hpctoolkit/install/bin/:$PATH
export PATH=/path/to/GVProf/install/bin/:$PATH
export PATH=/path/to/redshow/install/bin/:$PATH
```

Test if gvprof works.

```
cd ./samples/vectorAdd.f32
make
gvprof -e redundancy ./vectorAdd
hpcviewer gvprof-database
```

CHAPTER

THREE

MANUAL

3.1 Compile with Line Information

GVProf relies on debug information in binaries to attribute fine-grained value metrics on individual lines, loops, and functions.

For GPU binaries, we recommend using `-O3 -lineinfo`.

For CPU binaries, we recommend using `-O3 -g`.

For software compiled with CMake system, usually we can edit `CMAKE_C_FLAGS` and `CMAKE_CXX_FLAGS` to add line info flags. Additionally, CUDA line info can be added through `CMAKE_CUDA_FLAGS`.

3.2 Profile Using GVProf

The `gvprof` script automates a series of profiling and analysis processes, but supports only basic profiling features. For detailed profiling control, please refer to the next section.

```
gvprof -h
# Currently we offer three modes
# gvprof -v is your friend for debugging
gvprof -e <redundancy/data_flow/value_pattern> <app-name>
```

3.3 Profile Using HPCToolkit

Using hpctoolkit to profile applications enables fine-grained control knobs, selective analysis of GPU/CPU binaries, and compatibilities with various launchers (e.g., jsrun). We invoke `hpcrun` to profile an application twice using the same input. In the first pass, we dump the cubins loaded at runtime and profile each kernel's running time. Then we invoke `hpcstruct` to analyze program structure and instruction dependency. In the second pass, we instrument the cubins and invoke `redshow` redundancy analysis library to analyze measurement data.

3.3.1 First pass

```
hpcrun -e gpu=nvidia <app-name>
hpcstruct <app-name>
# if '--gpucfg yes', hpcstruct will analyze the control flow graph of each GPU function␣
↪and perform backward slicing, which is costly for large GPU binaries.
hpcstruct --gpucfg no hpctoolkit-<app-name>-measurements
# One can use also hpcstruct on the select GPU binaries only
hpcstruct --gpucfg no <binary-name>
```

3.3.2 Second pass

```
# Before profiling, we remove all profile data dumped in the first pass
rm -rf hpctoolkit-<app-name>-measurements/*.hpcrun

hpcrun -e gpu=nvidia,<mode> -ck <option1> -ck <option2> ... <app-name>
hpcprof -S <app-name>.hpcstruct hpctoolkit-<app-name>-measurements
# If only some binaries are analyzed using hpcstruct,
# one has to supply the corresponding binaries' structure files
hpcprof -S <app-name>.hpcstruct -S <binary-name>.hpcstruct hpctoolkit-<app-name>-
↪measurements
```

3.3.3 HPCToolkit separate pass

Large scale applications, such as Castro heavily use lambda functions and template functions for GPU kernels. Therefore, tools like `nsys` and `ncu` cannot efficiently correlate each kernel's execution time their names. Even though nvtx can provide some information to locate kernels, it is still not straightforward to map metrics back to source lines. Instead, we recommend using HPCToolkit, which provides an integrate calling context span CPUs and GPUs, to lookup the calling context and running time for each kernel. The following commands can be used.

```
hpcrun -e gpu=nvidia,pc <app-name>
hpcstruct <app-name>
hpcstruct --gpucfg no hpctoolkit-<app-name>-measurements
hpcprof -S <app-name>.hpcstruct hpctoolkit-<app-name>-measurements
hpcviewer hpctoolkit-<app-name>-measurements
```

3.4 Control Knobs

The following fine-grained options can be passed to either gvprof or hpcrun by pointing the option name and option value with `-ck <option-name>=<option-value>`.

```
HPCRUN_SANITIZER_GPU_PATCH_RECORD_NUM=<size of the buffer on GPU, default: 16 * 1024>
HPCRUN_SANITIZER_BUFFER_POOL_SIZE=<size of the buffer pool on CPU, default: 500>
HPCRUN_SANITIZER_APPROX_LEVEL=<enable approximated profiling, 0-5, default: 0>
HPCRUN_SANITIZER_PC_VIEWS=<number of top redundant values per pc, default: 0>
HPCRUN_SANITIZER_MEM_VIEWS=<number of top redundant values per memory object, default: 0>
HPCRUN_SANITIZER_DEFAULT_TYPE=<default data type of memory objects, default: float>
HPCRUN_SANITIZER_KERNEL_SAMPLING_FREQUENCY=<kernel sampling frequency, default: 1>
```

(continues on next page)

(continued from previous page)

```
HPCRUN_SANITIZER_WHITELIST=<functions to be monitored during execution, default: 0>
HPCRUN_SANITIZER_BLACKLIST=<functions not monitored during execution, default: 0>
HPCRUN_SANITIZER_READ_TRACE_IGNORE=<if read addresses are ignored, default: 0>
HPCRUN_SANITIZER_DATA_FLOW_HASH=<if SHA256 hash is calculated for every operation,␣
↪default: 0>
HPCRUN_SANITIZER_GPU_ANALYSIS_BLOCKS=<number of gpu blocks dedicated for analysis,␣
↪default: 0>
```

3.5 Interpret Profile Data

Currently, GVProf supports using hpcviewer to associate the redundancy metrics with individual GPU source code and using gviewer to process data flow metrics and prune unnecessary nodes/edges. We plan to integrate value pattern metrics into the data flow view for more friendly use of GVProf.

3.5.1 Calling context view

Only CPU calling context is available now. GPU calling context is under development.

```
hpcviewer <database-dir>
```

3.5.2 Data flow view

```
gviewer -f <database-dir>/data_flow.dot.context -cf file -pr
# gviewer -h for detailed options
```

The generated .svg can be visualized directly. To enable interactive control, we can rename the file to `demo.svg` and move it to `jquery.graphviz.svg`. After launch a server locally, we can visualize the graph, zoom in for important parts, and track each node's data flows.

3.5.3 Fine grain pattern views

```
# value pattern
less <database-dir>/value_pattern_t<cpu-thread-id>.csv

# redundancy
less <database-dir>/temporal_read_t<cpu-thread-id>.csv
less <database-dir>/temporal_write_t<cpu-thread-id>.csv
less <database-dir>/spatial_read_t<cpu-thread-id>.csv
less <database-dir>/spatial_write_t<cpu-thread-id>.csv
```

3.6 Example

CHAPTER

FOUR

FAQ

4.1 Profile Python applications

Please first refer to the MANUAL page for step-by-step profiling using HPCToolkit.

In addition to the basic commands there, we also have to pay attention to other minor issues.

In the measurement stage, `LD_LIBRARY_PATH=/path/to/python/library/:$LD_LIBRARY_PATH` may be needed as a prefix before *hpcrun*. We have a detailed example for profiling PyTorch.

Then, after getting measurement data and GPU binaries, we will analyze cpu binaries to get necessary line information. For GPU binaries, we use *hpcstruct –gpucfg no* on the measurement directory as suggested by the manual. For CPU binaries, the *python* binary does not contain all the program structure we need to understand program contexts. Instead, we have to analyze these binaries loaded dynamically at runtime. A python application may load hundreds of libraries at runtime but not use all of them. Therefore, in order to use hpcstrut on a minimum set of binaries but still extract information to understand program contexts, we adopt a *test-and-analyze* strategy. Using this strategy, we try hpcprof to correlate performance data with line maps first, if hpcprof hangs because of the large size of line map in a binary, we kill hpcprof and use hpcstruct on this binary to enjoy its fine-grained and fast binary analysis against hpcprof.

When hpcprof begins analyze a binary, it will print out some message like below. In such a case, we can kill hpcprof, remove the temporary database, and use `hpcstruct` to analyze `libtorch_python.so`.

```
msg: Begin analyzing : /path/to/python/lib/python3.8/site-packages/torch/lib/libtorch_
↪python.so
```

4.2 Accelerate data flow profiling

The following three knobs are helpful for accelerating proiling of applications with many kernels. With all the options turned on, the expected end-to-end of GVProf is approximately 20x, while the overhead could be over 1200x without these knobs.

```
HPCRUN_SANITIZER_READ_TRACE_IGNORE=<if read addresses are ignored, default: 0>
HPCRUN_SANITIZER_DATA_FLOW_HASH=<if SHA256 hash is calculated for every operation,␣
↪default: 0>
HPCRUN_SANITIZER_GPU_ANALYSIS_BLOCKS=<number of gpu blocks dedicated for analysis,␣
↪default: 0>
```

Note that these knobs can disable some information generation.

When GPU analysis is enabled, one can adjust the number of records on the GPU side to enlarge the buffer on the GPU side and further reduce overhead.

```
HPCRUN_SANITIZER_GPU_PATCH_RECORD_NUM=<size of the buffer on GPU, default: 16 * 1024>
```

4.3 Accelerate value pattern profiling

The following knobs are helpful for profiling the value pattern of specific kernels, focusing on just several kernel instances.

Besides, one can also apply `<pattern-name>@N` to activate block sampling that profiles a random GPU block out of every *N* blocks.

```
HPCRUN_SANITIZER_KERNEL_SAMPLING_FREQUENCY=<kernel sampling frequency, default: 1>
HPCRUN_SANITIZER_WHITELIST=<functions to be monitored during execution, default: 0>
HPCRUN_SANITIZER_BLACKLIST=<functions not monitored during execution, default: 0>
```

CHAPTER

FIVE

WORKFLOW

5.1 Use GPU Patch

GPU Patch is built upon Compute Sanitizer API. As we are closely working with NVIDIA on this API, we will update GPU Patch to use new features as soon as the new release is available. You can find a complete usage example of Sanitizer API in `sanitizer-api.c`. Some simple samples can be found in this repository.

5.2 Use RedShow with HPCToolkit

Please refer to the redshow header file for the complete set of interface.

If a new mode is added to GVProf, one should configure through the following redshow functions and sanitizer variables in HPCToolkit.

```
redshow_analysis_enable
redshow_output_dir_config

sanitizer_gpu_patch_type
sanitizer_gpu_patch_record_size
sanitizer_gpu_analysis_type
sanitizer_gpu_analysis_record_size
sanitizer_analysis_async
```

Currently, using a new runtime with redshow other than HPCToolkit is intricate, we will update the doc once we've gone through the whole process.

5.3 GVProf Tests

GVProf has end-to-end tests for each analysis mode plus an unit test for instruction analysis. Therefore, if a new analysis mode is added, we suppose the developer to add a test using python to verify its correctness.

For each analysis mode, the developer should write at least one simple case that covers most situations and collect results from samples.

We are in the process of completing the testing framework.

To run GVProf test, we use the following command at GVProf's root directory. The instruction test could fail due to the default data type used, which is acceptable.

```
python python/test.py -m all -a <gpu arch>
```

CHAPTER

SIX

ROADMAP

This document describes incoming features and release plans for GVProf. Since GVProf is a growing project, it has many components need fix and enhancement. Suggestions and feature requests are welcome. Users can post questions on Github's discussion forum.

6.1 Release v2.2

We plan release *v2.2* around Fall 2021, which will focus on enhancing the stability and compatibility of GVProf. Also, a few new features, such as customized memory allocator support and more accessible function filters are planned to be integrated.

- Features
  - NVTX

    Register CUPTI's NVTX callback to monitor customized memory allocators.
  - CUDA Memory Pool

    Support memory pool allocators in CUDA 11.2
- Bug Fixes
  - Function Filters

    Support substring match in whitelist and blacklist
  - Value Pattern Output

    Sort output arrays based on their access counts and fix weird numbers
- Deployment and Test
  - CMake

    Add CMake configurations to GVProf in addition to Makefile
  - Unittest

    Adapt python unittest package
  - Test configurations

    Adopt yaml files to configure test cases

6.2 Pending Issues

We haven't decided when to solve the following issues.

- GViewer Website

  Launch a website to visualize data flow graphs.

- Fine grain pattern and data flow integration

  Use the website described before to show both fine grain patterns and data flow.

- HPCToolkit Merge

  Merge the latest HPCToolkit master into GVProf.

CHAPTER

SEVEN

UNIT TESTS

7.1 interval_merge

This example is a carbon copy of the GVProf's interval analysis GPU module.

7.2 op_graph_simple

This example has a few redundant and duplicate memory access patterns, and is used to test basic functions of the GVProf's data_flow mode.

7.3 op_pattern_simple

This example has kernels with various fine-grained memory access patterns, and is used to test basic functions of the GVProf's value_pattern mode.

7.4 stress

A multi-context multi-stream proxy app to test GVProf's stability.

7.5 vectorAdd

A set of test cases for redshow's instruction parser.

CHAPTER

EIGHT

RODINIA GPU BENCHMARK

8.1 backprop

- vp-opt1: *value_pattern - single zeros*

`backprop_cuda_kernel.cu: 81`. The *delta* array has many zeros. We can check each entry on the GPU side to execute a special branch that avoid computation.

- vp-opt2: *data_flow - duplicate values*

`backprop_cuda.cu: 180`. *net->input_units* is copied to GPU at *Line 118* and copied back at *Line 188*. Meanwhile, both the GPU data and the CPU data are not changed. As a result, the copy at *Line 188* can be eliminated safely.

8.2 bfs

- vp-opt1: *value_pattern - type overuse*

`kernel.cu: 22`. The *g_cost*'s array's values are within the range of `[-127, 128)`. We can specify this array's type as `int_8` instead of `int` to reduce both kernel execution time and memory copy time.

- vp-opt2: *value_pattern - frequent values*

`bfs.cu: 107-109`. Accesses to these arrays showing a frequent value pattern where zeros are read most of the time. We can replace the memory copies of all zeros from CPU to GPU by memset that is much faster to reduce memory copy time.

8.3 cfd

- vp-opt1: *value_pattern - frequent values*

`euler3d.cu: 173`. The *cuda_initialize_variables* function writes values in a frequent value pattern. We can *hash* the accessing index of this array to limit memory access in a certain range and increase cache locality. Since this array is changed in the second iteration, this optimization only applies to the first iteration.

- vp-opt2: *data_flow - redundant values*

`euler3d.cu: 570`. The *old_variables* array is originally initialized at *Line 551* with the same values are *variables* but copied again at *Line 570*. We can safely eliminate the second copy which is redundant to the first iteration.

8.4 hotspot

- vp-opt: *value_pattern - approximate - single value*

`hotspot.cu: 164`. The *temp_src* array contains many very close floating point numbers. Using the approximate mode, gvprof determines values in this array are approximately the same under a certain approximation level. Therefore, we can read just some neighbor points on *Line 195* and still get similar final results.

8.5 hotspot3D

- vp-opt: *value_pattern - approximate - single value*

`opt1.cu: 29`. Like the *hotspot* example, the *tIn* array contains many very close floating point numbers. And gvprof determines all this values in this array are approximately the same under the certain approximation level. Incontrast to the *hotspot* example that selectively choose neighors, we use loop perforation to compute half of the loops and get similar result.

8.6 huffman

- vp-opt: *value_pattern - frequent values*

`his.cu: 51`. GVProf reports frequent values for the histo array in both the write and read modes. Because the most frequently updated value is zero, we can conditionally perform atomicAdd to reduce atomic operations.

8.7 lavaMD

- vp-opt: *value_pattern - type overuse*

`kernel_gpu_cuda.cu: 84`. The *rA* array contains only few distinct numbers. By checking its initialization on the CPU side, we note that there are only ten fixed values within 0.1 to 1.0. We can store these values using `uint_8` instead of `double`, saving *8x* space. These values are then decoded on the GPU side. In this way, we trade in compute time for memory copy time.

8.8 pathfinder

- vp-opt: *value_pattern - type overuse*

`pathfinder.cu: 144`. The *gpuWall* array's values for this input will be within `[0, 255]`, thereby we can use `uint8_t` to replace `int` to reduce global memory traffic.

8.9 srad

- vp-opt1: *value_pattern - heavy type*

`srad_kernel.cu: 79`. *d_c_loc* is always one or zero. We changed the value type of this array to bool.

- vp-opt2: *value_pattern - structured*

`srad_kernel.cu: 38` . *d_iN*, *d_iS*, *d_jW*, *d_jE* are used to indicate the adjacent nodes' coordinates which have structured patterns. We removed these four arrays and replace them with the corresponding calculations.

8.10 streamcluster

- vp-opt: *data_flow - redundant values*

`streamcluster_cuda.cu:221`. These arrays *center_table_d*, *switch_membership_d*, *p* are not changed in each iteration. Therefore, we can use flags on the CPU to detect if these arrays will be changed and only copy values if they are.

CHAPTER

NINE

QMCPACK

9.1 Introduction

QMCPACK is an open-source production level many-body ab initio Quantum Monte Carlo code for computing the electronic structure of atoms, molecules, and solids.

We study QMCPACK version `474062068a9f6348dbf7d55be7d1bd375c24f1fe`.

There are a bunch of packages required to compiled QMCPACK, including clang, OpenMP (offloading), HDF5, FFTW, and BOOST. These packages can be installed directly via spack.

To compile QMCPACK, we pass the following variables to cmake:

```
CMAKE_C_COMPILER=mpicc
CMAKE_CXX_COMPILER=mpicxx
ENABLE_OFFLOAD=ON
USE_OBJECT_TARGET=ON
OFFLOAD_ARCH=<gpu-arch>
ENABLE_CUDA=1
CUDA_ARCH=<gpu-arch>
CUDA_HOST_COMPILER=`which gcc`
QMC_DATA=<path/to/qmc/data>
ENABLE_TIMERS=1
```

The following environment variables are also required:

```
export OMPI_CC=clang
export OMPI_CXX=clang++
```

9.2 Profiling

First follow the instructions in `tests/performance/NiO/README` to enable and run the NiO tests. The configuration file used is `Nio-fcc-S1-dmc.xml` under the `batched_driver` folder.

At runtime, we use four worker threads (`export OMP_NUM_THREADS=4`). For a small scale run, one can adjust control variables such as `warmupSteps` to reduce execution time.

The data flow pattern can be profiled directly using gvprof. For the value pattern mode, one has to find the interesting function's names and use gvprof's whitelist to focus on these functions.

9.3 Optimization

- *data_flow - redundant values*

`MatrixDelayedUpdateCUDA.h: 627`. This line is often copying the same base pointers to the arrays on the GPU. Though this is not be a performance bottleneck for the current workload, it might be worth attention once the number of arrays increases.

CHAPTER

TEN

CASTRO

10.1 Introduction

Castro is an astrophysical radiation hydrodynamics simulation code based on AMReX framework.

We study Castro version `5e0a1b9cbc259f4dd17f5453ba59808b4da5c3ab`, and profile Casto's `Exec/hydro_tests/Sedov` example using its `inputs.2d.cyl_in_cartcoords` input.

To compile Castro, we setup the following variables in `GNUmakefile`:

```
USE_CUDA=TRUE
CUDA_ARCH=TRUE
DIM=2
USE_MPI=FALSE
```

10.2 Profiling

For a small scale run, we setup `max_step=20` in `inputs.2d.cyl_in_cartcoords`. To generate the data flow graph for Castro, along with redundancy metrics, we can use the `gvprof` script directly. For other fine-grained metrics, we can use `gvprof` if GPU control flow graphs are not required. Otherwise, we recommend using hpctoolkit to perform step-by-step profiling.

10.3 Optimization

- *data_flow - redundant values*

`AMReX_Interp_2D_C.H: 344`. When castro invokes `cellconslin_slopes_mmlim`, which is an internal function provided by AMReX, it performs `slope(i, j, n) *= a` for each output. With the `inputs.2d.cyl_in_cartcoords` input, somehow *a* is mostly 1.0. Thereby, we can save one load and one store for each output if we conditionally perform `slope(i, j, n) *= a`. Though this optimization does not achieve a significant speedup, it is worth mentioning if this it also benefits other applications that use AMReX.

CHAPTER

ELEVEN

DEEPWAVE

11.1 Introduction

Deepwave is a wave propagation software implemented based on.

We study deepwave version `1154692258da342accd21df02f7fa9ddd008f75f`. The input for deepwave is attached in GVProf's samples.

We first add `-lineinfo -g` to the `_make_cuda_extension` function in `setup.py`, and then add `-g` to the `_make_cpp_extension` function. Next we use `pip install .` to install deepwave.

Note that this pip is supposed be the pip installed by conda as we use conda across all the python samples

To run the deepwave example in GVProf, we need to install matplotlib by `conda install matplotlib`.

11.2 Profiling

Currently, using gvprof to profile python applications is intricate. We use HPCToolkit to profile and analyze deepwave separatedly. Please refer to the FAQ page for the complete guide.

With the default configuration, this example takes a relatively long time. We can change `num_epochs` to 1 and let it break after finishing the first batch. This deepwave application introduces higher overhead (150-200x) than other applications (~20x) because its kernels access millions of memory addresses with lots of gaps. As a result, we are not able to merge all of the memory access ranges on the GPU. Then, we will spend long time in both copying memory addresses from the GPU to the host and updating host memories.

For value pattern profiling, we monitor the most expensive `propagate` kernel using the following options.

```
LD_LIBRARY_PATH=/path/to/python/install/lib/python<version>/site-packages/torch:$LD_
↪LIBRARY_PATH hpcrun -e gpu=nvidia,value_pattern@10000 -ck HPCRUN_SANITIZER_WHITELIST=./
↪whitelist -ck HPCRUN_SANITIZER_KERNEL_SAMPLING_FREQUENCY=100000 python ./Deepwave_SEAM_
↪example1.py
```

For data flow profiling, we turn on these knobs to accelerate the profiling process.

```
LD_LIBRARY_PATH=/path/to/python/install/lib/python<version>/site-packages/torch:$LD_
↪LIBRARY_PATH hpcrun -e gpu=nvidia,data_flow -ck HPCRUN_SANITIZER_READ_TRACE_IGNORE=1 -
↪ck HPCRUN_SANITIZER_DATA_FLOW_HASH=0 -ck HPCRUN_SANITIZER_GPU_ANALYSIS_BLOCKS=1 -ck␣
↪HPCRUN_SANITIZER_GPU_PATCH_RECORD_NUM=131072 python ./Deepwave_SEAM_example1.py

# this gives you additional speedup
# export OMP_NUM_THREADS=16
```

More information about accelerating data flow and value pattern profiling can be found in the FAQ page

11.3 Optimization

Please refer to the `replication_pad3d` issue in PyTorch.

CHAPTER

TWELVE

DARKNET

12.1 Introduction

Darknet is an open source neural network framework written in C and CUDA. It is fast, easy to install, and supports CPU and GPU computation.

We check out Darknet version `312fd2e99a765949e468e18277d41f7992f08860`, study the `yolov4.cfg` and `yolov4-tiny.cfg` networks, and test an image `dog.jpg`.

To compile darknet, we setup the following knobs in Makefile:

```
GPU=1
# append -lineinfo to the start of ARCH
ARCH=-lineinfo ...
# append -g to the start of CFLAGS
CFLAGS=-g ...
```

12.2 Profiling

For the data flow analysis, one can use gvprof to profile darknet directly. `-ck HPCRUN_SANITIZER_READ_TRACE_IGNORE=1` yields significant speedup.

For the value pattern analysis, we recommend using a whitelist to specify interesting GPU kernels and turning on block sampling and kernel samples. In addition, if control flow graph based analysis is wanted, we don't recommend using `gvprof -cfg` directly because Darknet uses cuBLAS and cuDNN that trigger hundreds of large binaries loading at runtime. In fact, darkent's data type is almost uniform across all kernels so that one can gain insights even without `-cfg`.

We can profile the fine grain patterns of darknet using

```
gvprof -e value_pattern@10 -ck HPCRUN_SANITIZER_WHITELIST=./whitelist -ck HPCRUN_
↪SANITIZER_KERNEL_SAMPLING_FREQUENCY=20
```

In the `whitelist` file, we specify the following three kernels:

```
_Z15add_bias_kernelPfS_iiii
_Z21im2col_gpu_kernel_extiPKfiiiiiiiiiiiPf
_Z26activate_array_mish_kernelPfiS_S_
```

Other than a few kernels with frequent value patterns when approximation is used, we didn't find other interesting patterns.

You may want to lookup real kernel names with `gvprof -v` or `readelf -s` since compilers may generate different names

12.3 Optimization

- *data_flow - redundant values*

`upsampling_layer.c: 91` and `convolution_kernels.cu: 559`. In the generated data flow graph, we found that the nodes annotated with the `fill_ongpu` kernel always have redundant accesses. Because we run the inference mode only, the arrays are initialized with zeros and filled zeros again. To optimize it, we can set up a flag for each array to indicate if it is "clean". A "clean" array shouldn't be filled zeros again.

CHAPTER

THIRTEEN

PYTORCH

13.1 Introduction

PyTorch is a popular machine learning framework.

We use PyTorch version `f5788898a928cb2489926c1a5418c94c598c361b`. We study `resnet50`, `bert`, `deepwave` models.

We apply the following commands to compile PyTorch from source.

```
spack install miniconda3

conda install numpy ninja pyyaml mkl mkl-include setuptools cmake cffi typing_extensions␣
↪future six requests dataclasses

conda install -c pytorch magma-cuda110

export CMAKE_PREFIX_PATH=${CONDA_PREFIX:-"$(dirname $(which conda))/../"}
export USE_CUDA=1
export REL_WITH_DEB_INFO=1
export MAX_JOBS=16
export USE_NINJA=OFF
python setup.py install
```

- *resnet*

We get the `resnet` example from the pytorch benchmark repo.

To ease the installtion, we provide `1-spatial-convolution-model.py` and `1-spatial-convolution-unit.py` to check layer-wise and end-to-end performance.

- *deepwave*

We provide the instructions for installing deepwave here.

To ease checking the problematic kernel, we provide `2-replication-pad3d.py` script which only has a single `ReplicationPad3d` kernel.

- *bert*

We get the `reset` example from the pytorch benchmark.

To ease checking the problematic kernel, we provide `3-embedding-unit.py` script which only has a single `Embedding` kernel.

13.2 Profiling

Profiling a Python application takes extra steps than a normal application. We have a general guide to profile application in the FAQ page.

An example profiling command is attached below for reference:

```
LD_LIBRARY_PATH=/path/to/python/install/lib/python<version>/site-packages/torch:$LD_
↪LIBRARY_PATH hpcrun -e gpu=nvidia,data_flow -ck HPCRUN_SANITIZER_READ_TRACE_IGNORE=1 -
↪ck HPCRUN_SANITIZER_DATA_FLOW_HASH=0 -ck HPCRUN_SANITIZER_GPU_ANALYSIS_BLOCKS=1 -ck␣
↪HPCRUN_SANITIZER_GPU_PATCH_RECORD_NUM=131072 python ./<pytorch-script>.py
```

13.3 Optimization

We don't provide an automate performance testing suite for PyTorch in GVProf because recompile PyTorch for just small code changes still take long time and is a pain on low end servers.

- *data_flow - redundant values*

Please refer to this issue

- *data_flow - redundant values - value_pattern - single zeros*

Please refer to these two: issue1 and issue2

CHAPTER

FOURTEEN

NAMD

14.1 Introduction

NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems. NAMD uses the popular molecular graphics program VMD for simulation setup and trajectory analysis.

We download NAMD source code from its official website. We use NAMD version `4a41c6087f69c4cfe3edfdb19c6a5780ac20f5f1` and study the `alanin` input.

The following flags are setup in `Make.config`:

```
CUDAGENCODE = -arch <gpu-arch> -g -lineinfo
CXX_OPTS = -g -O3
```

14.2 Profiling

For data flow profiling, we use the normal gvprof script with the `-ck HPCRUN_SANITIZER_READ_TRACE_IGNORE=1` option.

For value pattern profiling, we monitor the most costly `nonbondedForceKernel` kernel of namd. Note that because this function accesses many arrays with different value types, we need GPU control flow graph and backward slicing to derive the types of each array. For your reference, we use the command

```
gvprof -cfg -j 16 -e value_pattern -ck HPCRUN_SANITIZER_WHITELIST=./whitelist -ck HPCRUN_
↪SANITIZER_KERNEL_SAMPLING_FREQUENCY=10
```

The CFG analysis phase could take up to an hour consuming about **100GB** main memory.

Caution: please use the full mangled name of `nonbondedForceKernel`

14.3 Optimization

- *data_flow - redundant values*

We find the *submitHalf* kernels are repetively invoked, forming an interesting diagram. Investigating carefully into the code, we find that the redundancy is introduced on purpose. The authors of namd pay close attention to its performance. They allocate some variables on the device to accumulate global sums and only transfer these value back to the host using the last block of the kernel. Besides, at the end of these kernels, they reset these values to zeros to make sure the next time the buffers are clean.

You may wonder they are doing this. There are two reasons:

1. If variables are not cleaned on the device, we have to reset variable using either `memsetAsync` or implicit device host communication which trigger extra cost. In contract, directly set variables in a GPU kernel can hide this latency by overlapping memory latency with computations latencies without additional API invocation.
2. If the host variable is accessed every time, these kernels will be slowed down significantly.

- *value_pattern - type overuse*

`CudaComputeNonbondedKernel.cu: 579`. By profiling the value patterns of this `CudaCOmputeNonbondedKernel` kernel, we find this array's type is overused. We can use `uint8_t` to replace the original `int` data type.

CHAPTER

FIFTEEN

BARRACUDA

15.1 Introduction

BarraCUDA is a GPU-accelerated sequence mapping software. BarraCUDA's code and sample data are open source and available at sourceforge. BarraCUDA's FAQ page provides useful instructions for installing and running benchmarks.

We study BarraCUDA *0.7.107h*, using the `Saccharomyces_cerevisiae.SGD1.01.50.dna_rm.toplevel.fa` sample data.

15.2 Profiling

The input we used elapses for a short time so that we can profile it directly using the `gvprof` script.

```
# prepare
./bin/barracuda index sample_data/Saccharomyces_cerevisiae.SGD1.01.50.dna_rm.toplevel.fa

# data_flow
gvprof -e data_flow ./bin/barracuda aln sample_data/Saccharomyces_cerevisiae.SGD1.01.50.
↪dna_rm.toplevel.fa sample_data/sample_reads.fastq > quicktest.sai

# value_pattern
gvprof -e value_pattern -cfg ./bin/barracuda aln sample_data/Saccharomyces_cerevisiae.
↪SGD1.01.50.dna_rm.toplevel.fa sample_data/sample_reads.fastq > quicktest.sai
```

15.3 Optimizations

- *data_flow - redundant values*

`barracuda.cu: 398`. In this function, cuda memory apis called after *Line 440* are not necessary when `number_of_sequences=0`. In that case, zero data are transferred between CPUs and GPUs such that arrays remain the same values, but still triggering API invocation cost.

- *value_pattern - frequent values*

`cuda2.cuh: 865`. This line copies all the elements from a local array to a global array, regardless of their values. While CPU's `memcpy` is fast for contiguous copy, GPU's `memcpy` is not. We observe that this copy operation involes many zeros. Therefore, we can create a `hits` array to record which positions have been updated, then only copy values at these positions.

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- search