Tensile Tuning and Workflow

William Gilmartin

Alex Liu

Scott Quiring

Babak Poursartip

June 26, 2024

Contents

1 Tensile and rocBLAS

rocBLAS is a BLAS implementation that is included as part of AMD's ROCm project offerings which optimizes the BLAS routines for AMD's GPUs. The GPU kernels which implement the rocBLAS blas3 gemm routines are optimized using Tensile, which is a utility that benchmarks the performance of selected kernels that are generated from "tuning parameters" and selected for inclusion in the final library based on the results. This library is included as part of the rocBLAS package and loaded at run-time during initialization. For a given gemm call, the problem parameters are passed to the Tensile client which selects the kernel with the best-known performance for the problem.

1.1 How to study this document

This document provides general information about the Tensile project. We recommend that a new reader clone and skim the Tensile repo before reading this document. Among the Tensile files, pay particular attention to

`<path-to-Tensile-repo>/Tensile/Common.py`

We will refer to this script in the following sections. We also recommend you visit the rocBLAS repo and its documentation, especially the *Level 3 BLAS GEMM* function, which is what Tensile is tuning for. A guide for the installation of rocBLAS and Tensile can be found in Appendix A.

1.2 Goals of Tensile Tuning

Every problem that can be solved with a generalized algorithm can be customized based on the application. In this case, we are solving problems on AMD GPUs which are highly configurable and can accomplish a given task in countless ways. We have many choices for distributing workload, hardware resources, vector parallelization, and unrolling loops, just to name a few. How do we know which are the best choices, guaranteeing correct results with the best performance?

One way that we can deal with this is by considering parameterized benchmarks. We choose parameters and problem sizes that we are interested in and try different combinations of these values to gain insight into their effects on correctness and performance. Once validated for the correctness and benchmarked for relative performance advantages, each of these combinations is considered a valid 'solution' to the problem. The goal of the Tensile tuning process is to build libraries containing GPU kernels that provide optimized performance for any problem specification within a given problem domain. The solution selection method maps an incoming problem or set of problems to the kernel which is optimal for the said problem(s).

1.2.1 Solution Selection Model

The model that Tensile uses for selection is a metric-based one that compares the target problem sizes against sizes used as representations (called exact sizes) within the model. These representation sizes are those tuned by Tensile during the tuning process so that for each exact size, there is a mapping from that specific size to the optimal kernel. In the model, if the incoming problem size happens to be one of the representative (exact) sizes, then the corresponding kernel is selected. However, if the problem size is not an exact size (a representation), then the selection falls back to an alternative means of selection, which in the current version of Tensile is based on a metric—or distance—from the representation space (that is, the set of exact sizes representing the model). The size specification is in the form of the gemm type problem description: based on the dimensions of the C matrix, the summation size, and the batch count of the problem. The metric selection finds the representative (exact) size which is closest to the incoming problem size in terms of the metric. The selection then returns the kernel which is optimal for the representative size.

The objective of building the Tensile library is to find the optimal set of representative sizes and associated kernels which best models the problem domain. The Tensile tuning process will take a given set of problem sizes and create the internal mappings which map those problem sizes to the respective optimal kernel.

1.2.2 Tuning with Tensile

The Tensile tuning process performs tuning for sets of problem sizes (the problem space) upon sets of kernels (the kernel space) defined by the specification of the tuning. The set of problem sizes along with the set of kernels together defines the configuration space for the tuning:

$$\text{tuning configurations space} = \text{solutions} \times \text{problem sizes}.$$

The results of the tuning will produce the ideal kernel from within the kernel space for each problem size within the problem space, thus generating the mapping which is used in the selection.

2 Basic Example (Hello Tensile Tuning)

As a simple example of running the tuning application, look at the configuration file tuning.yaml. This provides Tensile with the specifications for a simple tuning run using Tensile: the content will be described in more detail latter in this document. The following example shows how to execute a full round of tuning using Tensile.

```
# clone Tensile
$ git clone https://github.com/ROCmSoftwarePlatform/Tensile.git

# set up tuning run
$ cd Tensile
$ mkdir build_test
# we perform all the tuning processes in a folder starting with build_*. Any folder
    like this will be ignored in the repo.
$ cd build_test
# Tensile/tuning_docs/examples has some sample Tensile input configuration file. See
    also Tensile/Tensile/Configs
$ cp ../tuning_docs/examples/example_vega20_tuning.yaml .

# run tensile
$ ../Tensile/bin/Tensile example_vega20_tuning.yaml . > tuning.out 2>&1
```

In this example, we ran the tuning configuration for a system with a vega20 GPU installed (this example configuration can be found in the Tensile/tuning_docs/examples sub directory of the Tensile repo). The configuration parameters which enable tuning for this specific device are described in the Library Logic Parameters section of this document.

The output of the benchmark run (tuning.out) contains useful information for examining the performance of the problem sizes as measured against the selected set of kernels that are being benchmarked. An excerpt is shown below.

```
run, problem-progress, solution-progress, operation, problem-sizes, solution,
 validation, time-us, gflops, empty, total-gran, tiles-per-cu, num-cus, tile0-gran,
 tile1-gran, cu-gran, wave-gran, mem-read-bytes, mem-write-bytes, temp-edge,
 clock-sys, clock-soc, clock-mem, fan-rpm, hardware-samples, enqueue-time

0, 0/3, 0/5, Contraction_l_Ailk_Bljk_Cijk_Dijk, "5504,5504,1,3104",
 Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU1_K1_TT4_4_WG16_16_1, NO_CHECK, 17423.79,
 10794, , 0.99, 123.27, 60, 1.00, 1.00, 0.99, 1.00, 11875254272, 121176064,  36.00,
 700.00, 971.00, 1000.00, 845.00, 1, 2020-09-29 13:32:14.002548

0, 0/3, 1/5, Contraction_l_Ailk_Bljk_Cijk_Dijk, "5504,5504,1,3104",
 Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU4_K1_TT4_4_WG16_16_1, NO_CHECK, 17953.34,
 10475, , 1.00, 493.07, 60, 1.00, 1.00, 1.00, 1.00, 11754078208, 363528192, 36.00,
 1801.00, 971.00, 1000.00, 845.00, 1, 2020-09-29 13:32:14.029611

0, 0/3, 2/5, Contraction_l_Ailk_Bljk_Cijk_Dijk, "5504,5504,1,3104",
 Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU1_K1_TT8_8_WG16_16_1, NO_CHECK, 15872.19,
 11849, 0.99, 30.82, 60, 1.00, 1.00, 0.99, 1.00, 5998215168, 121176064, 36.00,
 1801.00, 971.00, 1000.00, 845.00, 1, 2020-09-29 13:32:14.058389

0, 0/3, 3/5, Contraction_l_Ailk_Bljk_Cijk_Dijk, "5504,5504,1,3104",
 Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU4_K1_TT8_8_WG16_16_1, NO_CHECK, 17040.37,
 11036, 0.99, 123.27, 60, 1.00, 1.00, 0.99, 1.00, 5877039104, 363528192, 36.00,
 1801.00, 971.00, 1000.00, 845.00, 1, 2020-09-29 13:32:14.082752

0, 0/3, 4/5, Contraction_l_Ailk_Bljk_Cijk_Dijk, "5504,5504,1,3104",
 Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU1_K1_TT8_8_WG8_8_4, NO_CHECK, 26010.67,
 7230, 1.00, 493.07, 60, 1.00, 1.00, 1.00, 1.00, 11875254272, 121176064, 36.00,
 1801.00, 971.00, 1000.00, 843.00, 1, 2020-09-29 13:32:14.110139
```

```
. . .
```

The text in the red shows the problem dimensions, the kernel name, and the performance (GFlops) as measured by Tensile (to quickly detect GFlops in each line, look for ,,.). This is useful when iterating through a tuning exercise that takes more than one benchmark run. The output also provides the user with a snapshot of the performance of the sizes for each of the kernels being measured by Tensile. In this example, the blue text indicates the progress of the problems and solutions. A problem is defined by the size specification (the sizes that are being benchmarked) and the solutions are defined by the kernels generated by Tensile during the initial stages of execution. Here, we see problem 0 of 3 (0/3) is being presented. This problem was benchmarked for each of the kernels that Tensile generated, in this case, 5 total kernels ([0-4]/5). The performance of the first measurement is highlighted in red. The final results of the tuning will be the best performing kernel for each of the sizes being measured; in this example, it is highlighted in green. We can see here that the best kernel for problem 0 was Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU1_K1_TT8_8_WG16_16_1, and hence, this is the one that will be selected for the final library.

As part of the tuning process, one can examine these results and get a sense of how good the tuning parameters are that are being used in the current sweep of the tuning parameters. If needed, adjustments to the tuning parameters can be made for later iterations.

3   Tensile Tuning Workflow

The Tensile tuning workflow may be viewed conceptually as a different operation than a simple Tensile tuning pass, which performs tuning as a single pass of the Tensile application. During a complete tuning exercise, it could take multiple iterations to reach the desired performance. A single tuning iteration of tuning in Tensile can be broken down into several steps; the Tensile application performs those steps as part of a complete unit.
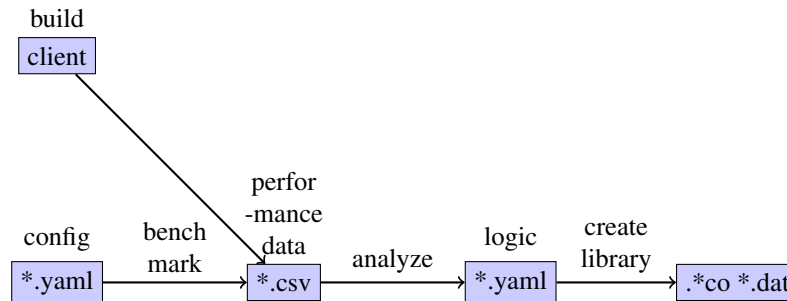
3.1   Steps in Tensile Tuning

Figure 1: a complete Tensile tuning pass

The input to Tensile is a tuning configuration file that provides the set of parameters used to drive the tuning process for a given iteration. Before running Tensile, the user should prepare and drop the configuration (*.yaml) file in the working directory (build*). Upon running Tensile, it reads the configuration file and performs the steps indicated in Figure 1. Each of the steps corresponds to the directories created by the Tensile application located within the working directory:

Step 0 This step creates the Tensile client application which is the executable that does the benchmarking. This is found in the directory **0_Build**.

Step 1 The second step does two tasks. The first part of this step generates a library containing all the solutions that are being measured based on the tuning configuration. The second part of this step does the actual benchmarking using the problems sizes defined in the configuration. The output of this is placed in **1_BenchmarkProblems**.

Step 2 This step merely collects and copies the benchmark data and places it in **2_BenchmarkData**.

Step 3 This step performs analytics on the data and generates the logic file which is used to create the final Tensile Library. The output of this is placed in **3_LibraryLogic**. There will be one yaml file for each type of the problems defined in the configuration input file, each of which contains the solution information for all sizes for said problem type. This directory will be generated only if we define LibraryLogic section in the configuration file. Refer to section 3.3.3.

Step 4 Finally, this step generates a prototype Tensile Library based on the logic file created in the previous step. The output of this is placed in **4_LibraryClient**. This directory will be generated only if we define LibraryClient in the configuration file. Refer to section 3.3.4.

The full workflow also includes post-process analysis of the results. In our current system, we use rocblas-bench (will be discussed in 4.1) which includes a reference version of the library that can be used to compare the results. That is, we make a comparison of the version of rocBLAS without the updated tuning and a new candidate version which includes the new tuning results.

The overall process includes user intervention in several areas of the tuning workflow, which includes the creation and modification of the tuning configuration for each iteration of the tuning process. The automation helps in the process by automating the initial tuning and validation part of the workflow, but it is not smart enough to update parameters sets for additional steps in the process.

3.2 Tensile Application

The tensile tuning application is executed using the following command:

```
./Tensile/bin/Tensile config_file output_path

arguments:
 config_file            benchmark config.yaml file
 output_path            path where to conduct benchmark
```

Tensile takes a tuning configuration file tuning_config.yaml as input. This file provides all the specifications of the tuning, including the global context—such as the print level and build type (Release—Debug)—as well as the sets of parameters used to specify the problem, sizes, and definition of the kernels used for benchmarking. Tensile will produce, as a result of the tuning, a Cijk_Ailk_Bljk.yaml file (called the logic file, dropped in 3_LibraryLogic directory) which contains all the information needed to build the final Tensile library, which includes the kernel objects and the definitions of the mappings which map a given input to the optimized solution for that input. Next section (3.3) explains how to prepare and customize the configuration file.

3.3 The Tensile Tuning Configuration (input to Tensile)

The Tensile tuning configuration defines the context which drives the tuning for a complete or partial tuning pass. There are *4 sections* defined in this configuration which do *not* directly correspond to the directories defined in the previous section (3.1). The sections are

GlobalParameters The set of parameters that provides context for the entire tuning exercise.

BenchmarkProblems Defines the set of kernel specifications as well as the size definitions for the tuning exercise.

LibraryLogic Specifies the target environment and platform information that Tensile uses as part of the library logic. The logic files that get generated provide Tensile with the metadata used in the final Tensile library construction.

LibraryCilent If defined, this will enable step 4 of the tuning process which means the final library will be created. There are no parameters that are defined in this section, so this is just a flag that indicates that the final library is to be built. The library generator will take, as input, the logic files created in the previous step.

3.3.1 GlobalParameters

Below is an example of the global parameters section in the tuning configuration.

```
1  GlobalParameters:
2    MinimumRequiredVersion: 4.4.0
3    PrintLevel: 1
4    ForceRedoBenchmarkProblems: True
5    ForceRedoLibraryLogic: True
6    ForceRedoLibraryClient: True
7    CMakeBuildType: Release
8    EnqueuesPerSync: 1
9    SyncsPerBenchmark: 1
```

```
10    LibraryPrintDebug: False
11    NumElementsToValidate: 0
12    ValidationMaxToPrint: 4
13    ValidationPrintValids: False
14    ShortNames: False
15    MergeFiles: True
16    Platform: 0
17    Device: 0
18    KernelTime: True
19    PinClocks: False
20    SleepPercent: 0
21    DataInitTypeBeta : 0
22    PrintWinnersOnly: 1
23    PrintSolutionRejectionReason: False
24    DataInitTypeA: 1
25    DataInitTypeB: 8
26    DataInitTypeC: 0
27    DataInitTypeD: 0
28    DataInitTypeBeta: 2
29    DataInitTypeAlpha: 2
30    PrintTensorA: 0
31    PrintTensorB: 0
32    PrintTensorC: 0
33    PrintTensorD: 0
```

Some important parameters to consider are the following:

CMakeBuildType Specifies the build type of the tensile client: set to either Release or Debug

PrintLevel The print level. Possible choices: 0=none, 1=standard, 2=verbose.

Device If multiple devices are installed on the machine used for tuning, this will indicate the device id used to perform the benchmarks.

NumElementsToValidate Validates the kernel output with CPU results. The higher number of elements results in a slower tuning process.

PrintSolutionRejectionReason The default is False. Not all solutions created from the parameters are valid. Setting this to True prints out the reason for rejecting the solution. SolutionStructs.py generates/rejects the solutions.

DataInitTypeA/B/C/D Defines initialization methods for the tensors. Good for debugging. See Common.py for details.

DataInitTypeBeta/Alpha Defines initialization values for alpha and beta. Good for debugging. See Common.py for details.

PrintTensorA/B/C/D Prints the tensors in the output. Default is 0. Set to 1 for debugging.

See Common.py in Tensile repo for the full list of input parameters.

3.3.2 BenchmarkProblems

```
1    BenchmarkProblems:
2    - - {Batched: true, DataType: s, OperationType: GEMM, TransposeA: false,
     ↪ TransposeB: false, UseBeta: true}
3      - BenchmarkCommonParameters:
4        - LoopTail: [true]
5        - KernelLanguage: [Assembly]
6        - EdgeType: [ShiftPtr]
7        BenchmarkFinalParameters:
8        - ProblemSizes:
9          - Exact: [ 5504, 5504, 1, 3104 ]
```

```
10      - Range: [ [64, 64, 64, 7000], [4], [1], [128] ]
11      BenchmarkForkParameters: null
12      BenchmarkJoinParameters: null
13      ForkParameters:
14      - GlobalSplitU: [1, 4]
15      - PrefetchGlobalRead: [true, false]
16      - WorkGroupMapping: [1, 8, 64]
17      - DepthU: [16, 32, 64]
18      - PrefetchLocalRead: [true, false]
19      - VectorWidth: [2, 4]
20      - GlobalReadVectorWidth: [1, 4]
21      - FractionalLoad: [0, 1]
22 # the thread tilings
23      - ThreadTile:
24        - [4, 4]
25        - [8, 8]
26 # the work group specifications
27      - WorkGroup:
28        - [16, 16, 1]
29        - [8, 8, 4]
30      InitialSolutionParameters: null
31      JoinParameters: null
32
33 # can define more than 1 problem in a problem group
34 # start of next problem
35    - BenchmarkCommonParameters:
36
37 ...
38
39 # other problem groups
40 - - {Batched: true, DataType: s, OperationType: GEMM, TransposeA: false,
↪    TransposeB: false, UseBeta: true}
41
42 ...
43
```

This is the *most important section* of the tuning configuration; it defines specifications for the set of kernels that are being benchmarked as well as the sizes that are being targeted. Each benchmark problem has a set of parameters that define how everything gets constructed. Most of the parameters are descriptions of the kernel specifications. The main parameters to focus on for generating the benchmark kernels are BenchmarkCommonParameters and ForkParameters; they get generated in the following convention.

BenchmarkCommonParameters The benchmark common parameters define a base set of parameters used in all the kernels that are generated in the specification for the section. These would then be common parameter values for all the kernels that are generated.

ForkParameters The fork parameters generate the kernels as a Cartesian product of the parameters lists which are specified. For example, if VectorWidth = [2, 4] and DepthU = [16, 32], the kernels that get generated would be drawn from all combinations of the corresponding value lists. Hence, the combinations { {VectorWidth=2 ,DepthU=16}, {VectorWidth=4, DepthU=16}, {VectorWidth=2, DepthU=32}, {VectorWidth=4, DepthU=32} } would get generated as part of the respective kernels specifications, and they would be targeted for benchmarking provided they define valid kernels.

BenchmarkFinalParameters The problem sizes get defined in BenchmarkFinalParameters as shown in the example provided. There are two types of size specifications: one is the Exact, which specifics a single size, and the other is Range, which generates a set of sizes based on the range logic.

Note the overall structure of this section of the configuration:

Benchmark Problem Groups (List of problem groups):
    – Problem Group (List of problem groups items)
        – Problem (problem group items)
            – Problem Type (first item)
            – Problem One Spec (second item (parameter map))
            – Problem Two Spec (third item (parameter map))
            – Problem Three Spec (fourth item (parameter map))
    ...
    BenchmarkForkParameters, JoinParameters, BenchmarkJoinParameters are obsolete items in this section.

3.3.3 Library Logic

The Library Logic is a collection of one or more yaml files containing benchmarking output results (usually one per gfx architecture, per benchmark problem) created by Tensile and is the final product of the benchmarking and tuning process (dropped in 3_LibraryLogic). It consists of a list of solutions, a list of problem sizes, and mappings from each problem size to the fastest solution. If we skip this section in the configuration file, the library logic file (and 3_LibraryLogic directory) will not be generated.

This section of the tuning configuration specifies the target platform and architecture of the particular tuning exercise being conducted. An example of what gets defined in this section is:

```
1  LibraryLogic:
2    ArchitectureName: gfx906
3    DeviceNames: [Device 66a0, Device 66a1, Device 66a7]
4    ScheduleName: vega20
```

This describes the target platform for which the final library is targeted and will generate a corresponding logic file for this specific platform and device specification.

There are three main parts to this and they are mostly self-describing. The ArchitectureName specifies the graphics architecture and the number "906" is the instruction set "ISA" id. The schedule is the GPU codename. The Device list is the specific device ids. Other example target platform specifications and devices are listed in Appendix D.

Generated Logic File Example:
Notice that the top of this logic file contains the same information as described in the LibraryLogic section of the configuration file. The content of this file will be discussed in section 3.5.2

```
1  # specifications of library
2  # defined in GlobalParameters section of the tuning configuration
3  - {MinimumRequiredVersion: 4.22.0}
4  # defined in the LibraryLogic section of the tuning configuration
5  - vega20
6  - gfx906
7  - [Device 66a0, Device 66a1, Device 66a7]
```

3.3.4 LibraryClient

The LibraryClient section is optional. Including it in the tuning config will build the Tensile library from the library logic files after tuning finishes and then runs the fastest solution for each tuned size.

```
1  #LibraryLogic:
```

3.4 Tuning Configuration Construction

There are two main components to focus on during the construction of the tuning configuration: the problem type and the list of problem specifications. The problem type, shown below, is fairly straightforward.

```
- - { Batched: true,         # [true, false]
      DataType: s,           # [S=singel, D=Double, C=ComplexSingle,
  ↪   Z=ComplexDouble, H=half,4xi8=int8x4, I=int32, B=bfloat16]
      OperationType: GEMM,   # [GEMM, TensorContraction, ConvolutionForward,
  ↪   ConvolutionBackwardData, ConvolutionBackwardWeights]
      TransposeA: false,     # [true, false]
      TransposeB: false,     # [true, false]
      UseBeta: true          # [true, false]
    }

```

The list of problem specifications is the heart of the tuning configuration. Tensile has the ability to define a multi-step tuning process that includes a complex set of search specifications, but in practice does not generally result in better performance than the process we outline here.

```
BenchmarkCommonParameters:
BenchmarkFinalParameters:
BenchmarkForkParameters: null       # not used
BenchmarkJoinParameters: null       # not used
ForkParameters:
InitialSolutionParameters: null     # not used
JoinParameters: null                # not used
```

The **BenchmarkCommonParameters** and **ForkParameters** define the space of kernels used for benchmarking. The benchmark common parameters define the common parameters that—unless overridden by the forked parameters—are defined for all the kernels that get generated for the benchmarking. As described in the forked parameters section, the fork parameters define a Cartesian product of tuning parameter created from all the combinations of parameters in the parameter lists. Here we will focus on the most common parameters that are used during most tuning exercises.

ThreadTile = [tt0, tt1] and **WorkGroup** = [wg0, wg1, localSplitU] define the thread tile parameter which defines the dimensions of the C matrix that each thread works on. Together they form the macro tiling **MacroTile** = [tt0*wg0, tt1*wg1, mt2]. The **GlobalSplitU** parameter defines how Tensile splits the unroll summation into multiple sections. Other possible considerations for starting the tuning are **DepthU, VectorWidth** and **GlobalReadVectorWidth**. The global read vector width Controls desired width (number of elements) for loads from global memory LDS (possable values are [-1, 1, 2, 3, 4, 6, 8]). The vector width controls the contiguous elements from the C matrix. The depth U parameter is closely related to the local split U parameter and the combination of the two defines how the loop of the summation dimension is unrolled.

Considering the full set of parameters involved, it is not practical to build every set of kernels as part of any investigation; this would exhaust compute resources. The target number of kernels we consider optimal for investigation is between 10,000 and 20,000. The tuning workflow may take many iterations of the tuning to reach optimal performance. Each iteration will perform a complete run of the Tensile tuning and evaluation of the results. If some of the problem sizes under investigation do not result in optimal performance then, for the sizes that fail, modifications of the tuning specifications are made to search for alternative solutions in the kernel space. The tuning is then rerun using the updated parameter set. This type of iteration is continued until the desired optimization is reached.

3.5 Tuning Artifacts

The final result of Tensile tuning will generate some data artifacts which are important to know about.

3.5.1 Build Artifacts

The first stage of the tuning creates a benchmark library. This is found in the directory:

```
$ ls /build_test/1_BenchmarkProblems/Cijk_A??k_B??k_??_??/??_Final

# sourcs contains the library components
$ build source

$ ls /build_test/1_BenchmarkProblems/Cijk_A??k_B??k_??_??/??_Final/source

# library contains the final tensile library and code objects
# build_tmp contains the kernals
$ build_tmp library

$ ls /build_test/1_BenchmarkProblems/Cijk_A??k_B??k_??_??/??_Final/source/build_tmp/
    SOURCETMP/assembly

$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU4_K1_TT8_8_WG8_8_4.co
$ Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU1_K1_TT8_8_WG16_16_1.o
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU1_K1_TT4_4_WG16_16_1.co
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU1_K1_TT8_8_WG8_8_4.s
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU4_K1_TT8_8_WG8_8_4.o
$ Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU1_K1_TT8_8_WG16_16_1.s
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU1_K1_TT4_4_WG16_16_1.o
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU4_K1_TT4_4_WG16_16_1.co
$ Cijk_Ailk_Bljk_SB_MT64x64x16_SE_GSU4_K1_TT8_8_WG8_8_4.s

$ ls /build_test/1_BenchmarkProblems/Cijk_A??k_B??k_??_??/??_Final/source/library

$ Kernels.so-000-gfx1010.hsaco  Kernels.so-000-gfx1011.hsaco Kernels.so-000-gfx803.
    hsaco Kernels.so-000-gfx900.hsaco Kernels.so-000-gfx906.hsaco  Kernels.so-000-
    gfx908.hsaco TensileLibrary_gfx906.co TensileLibrary.yaml
```

Here, the question marks indicate that there may be more than one subdirectory created for each level, due to the fact that a configuration can specify more than one problem type or problem groupings. The naming convention corresponding to Cijk_A??k_B??k_??_?? is a result of the problem type. The letters C, A and B are a reference to the matrix multiplication $C = A \times B$; in this context, the letters convey the summation index $= l$, batch index $= k$ and the fixed indices $= i, j$. Hence,

Cijk_Ailk_Bljk $\implies C = A * B$

Cijk_Ailk_Bjlk $\implies C = A * B^T$

Cijk_Alik_Bljk $\implies C = A^T * B$

Cijk_Alik_Bjlk $\implies C = A^T * B^T$

The names for the problem types are effectively the Einstein summation convention for tensor contraction. This convention is that, in a summation over symbols that appear twice, the summation is simplified by making the summation symbol implicit.

$$C_{ijk} = \sum_{l=1}^{N} A_{ilk} B_{ljk} = A_{ilk} B_{ljk} \text{ (Einstein: the sum is over the index } l)$$

The other part of the name corresponds to the datatype. Additionally, if there is more than one problem in the problem group defined in the configuration, there will be a sequence number attached. Hence, Cijk_Ailk_Bljk_SB_00 means that the problem is the first problem in the problem group and is for datatype single. The directory under this could possibly contain more than one subdirectory which corresponds to the sub-steps in the configuration for the specific problem; however, in practice, we recommend following the conventions specified above which will only produce one subdirectory for one tuning step. The **00_Final** directory contains the resulting library which is used by Tensile to perform the actual benchmarks.

The files in the **source/library** directory include the compiled shared library and the serialized master library. The code objects are identified by the extensions .co/.hsaco. The only difference between them is that the .co libraries contain the ASM (assembly) kernels and the .hsaco libraries contain SOURCE kernels. The gfx??? part of the name identifies the architecture for which the objects were compiled for.

The full kernel naming conventions is outlined in Appendix C.

3.5.2 The Tensile Logic File

The result of tuning is the Tensile library logic file(s) in build*/3_LibraryLogic. These files are used to generate the final Tensile library using the TensileCreateLibrary application. This is a separate application (still in the Tensile repo) that takes the library logic files generated by Tensile as input. This application will be discussed further in section 5. Here is a sample logic file:

```
1   # specifications of library
2   # defined in GlobalParameters section of the tuning configuration
3   - {MinimumRequiredVersion: 4.22.0}
4   # defined in the LibraryLogic section of the tuning configuration
5   - vega20
6   - gfx906
7   - [Device 66a0, Device 66a1, Device 66a7]
8   - AllowNoFreeDims: false
9     AssignedDerivedParameters: true
10    Batched: true
11    ComplexConjugateA: false
12    ComplexConjugateB: false
13    ComputeDataType: 0
14    ConvolutionConfig: []
15    DataType: 0
16    DestDataType: 0
17    HighPrecisionAccumulate: false
18    Index0: 0
19    Index01A: 0
20
21  ...
22
23  # first kernel
24  - - 1LDSBuffer: 0
25      AggressivePerfMode: 1
26      AssertFree0ElementMultiple: 1
27      AssertFree1ElementMultiple: 1
28      AssertMinApproxSize: 3
29  ...
30
31      ProblemType:
32        AllowNoFreeDims: false
33        AssignedDerivedParameters: true
34
35  ...
36
37      ThreadTile: [4, 4]
38      WorkGroup: [16, 16, 1]
39
40  ...
41
42  # start of the next kernel
43    - 1LDSBuffer: 0
44
45
46  # start of the size mappings
47  - [2, 3, 0, 1]
48  # size mapping
49  # size [m, n, batch, k, strides, ...]
50  - - - [64, 4, 1, 128, 64, 64, 64, 128]
51  # mapping of size to kernel 0
52      - [0, 3.0]
```

12

```
53      - - [128,  4,  1, 128, 128, 128, 128, 128]
54        - [0,  6.0]
55    # mapping of size to kernel 1
56      - - [256,  4,  1, 128, 256, 256, 256, 128]
57        - [1, 13.0]
58
59    # ignored in the current selection model
60    - null
```

This example only highlights what the logic file looks like. It is not practical, at this point, to go over all the parameters in a complete file. There are, however, two parts worth noting at the moment: near the beginning of the file is the library specifications that was defined in the LibraryLogic section of the tuning configuration; near the end of the file are the mappings.

The mappings in the logic file are specified in pairs; the first array specifies the size and the second specifics what kernel that size is mapped to. The size part is parameterized as follows (in terms of BLAS parameters):

$$- - [m, n, \text{batch size}, k, ldc, ldd, lda, ldb].$$

The second array gives the index of the kernel this size is mapped to as the first parameter and the measured efficiency of the selected kernel as the second:

$$- [\text{kernel index, measured efficiency}].$$

Note that as of now, rocBLAS does not store the ld (leading dimension) information in the final library, even if we define them in the configure file.

Naming convention:

Tensile generates one library logic file for each problem type. They are named as follows:

$$\langle architecture\_name \rangle \_ \langle Contraction\_Type \rangle \_ \langle Data\_Type \rangle$$

$\langle architecture\_name \rangle$ refers to the GPU architecture, for example for Vega 20, the architecture name is vega20. $\langle Contraction\_Type \rangle$ refers to the Einstein tensor convention as discussed in section 3.5.1. $\langle Data\_Type \rangle$ is based on the ProblemType in the configuration file. For example, SB refers to a Batched Single precision DataType. HSS_BH means Half-precision for DataType (A/B), Single-precision for DestDataType (C/D), and Single-precision for ComputeDataType. Notice also that the exact same naming convention will be used in the rocBLAS library.

4 Integrating new libraries in rocBLAS

Once we have library logic files for new sizes, we need to merge and update rocBLAS libraries. This is a two-fold process:

- First, we need to confirm that the new logic file is generating the correct results and the performance of the new library is more than the old library. We use rocblas-bench tool for this part. See the next section.

- Merging the new logic file into rocBLAS using the merge tool (merge.py in Tensile). See section 4.2.

4.1 rocblas-bench tool

rocblas-bench is a tool used to perform benchmarks on the problem sizes which are being measured and to verify the correctness of the new libraries. While it is designed to measure the performance of the full set of rocBLAS library functions, our interest is in the level 3 gemm functions which—in rocBLAS— are called into the Tensile library.

The git repository for rocBLAS is found at https://github.com/ROCmSoftwarePlatform/rocBLAS. Either the develop or master branch version may be used for testing.

```
$ git clone -b master|develop https://github.com/ROCmSoftwarePlatform/rocBLAS.git
```

One method to use rocblas-bench is to install it locally. The library can be installed using the included installation script (the installation may last quite some time.):

```
$ cd rocBLAS
$ ./install.sh -h
$ ./install.sh -d -c -a <architecture> --cmake_install
```

where the c option builds the client applications, which includes rocBLAS-bench, the d option builds and installs the dependencies (only needed to be included once on each system that rocBLAS is being built), and the a option specifies the GPU architecture e.g. gfx906:xnack-, gfx908:xnack-. Without this flag, rocBLAS installs for all architectures, and installation lasts even longer. The cmake flag allows the rocBLAS to install cmake if it is older than required.

Executing the test for a single problem size is done using familiar blas parameters:

```
rocBLAS$ cd build/release/client/staging
rocBLAS/build/release/client/staging$ ./rocblas-bench -h
rocBLAS/build/release/client/staging$ ./rocblas-bench -f gemm -r f32_r --transposeA
    N --transposeB T -m 1001 -n 1536 -k 64 --alpha 1.0 --lda 1001 --ldb 1536 --beta
    0.0 --ldc 1001 -i 10
rocBLAS/build/release/client/staging$ ./rocblas-bench -f gemm_ex --transposeA N --
    transposeB T -m 1024 -n 1024-k 1024 --a_type h --lda 2048 --b_type h --ldb 2048
    --beta 1 --c_type s --ldc 2048 --d_type s --ldd 2048 --compute_type s -v 1 --
    iters 10 --cold_iters 2
```

Here, the i/iters option is the number of iterations of the gemm call. See the help output for other flags. Because there is a latency in loading the Tensile library to which rocBLAS links, for testing, it is recommended to use rocblas-bench with the batch call. The batch call takes a yaml file with a list of problem sizes. An example yaml file contents is shown below (problem-size.yaml).

```
1
2  - { rocblas_function: "rocblas_sgemm", transA: 'N', transB: 'N', M: 1024, N: 1264,
   ↪  K: 345, lda: 1024, ldb: 345, ldc: 1024, cold_iters: 2, iters: 10 }
3  - { rocblas_function: "rocblas_sgemm", transA: 'N', transB: 'N', M: 1025, N: 1064,
   ↪  K: 148, lda: 1025, ldb: 148, ldc: 1025, cold_iters: 2, iters: 10 }
4  - { rocblas_function: "rocblas_gemm_ex", transA: "N", transB: "T", a_type: f16_r,
   ↪  b_type: f16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M:    4096,
   ↪  N:    4096, K:    1024, lda:    8192, ldb:    8192, ldc:    8192, ldd:    8192,
   ↪  cold_iters: 2, iters: 10, beta: 1 }
5  - {rocblas_function: rocblas_gemm_ex, transA: N, transB: T, a_type: f16_r, b_type:
   ↪  f16_r, c_type: f32_r, d_type: f32_r, compute_type: f32_r, M: 1024, N: 1024, K:
   ↪  1024, lda: 8192, ldb: 8192, ldc: 8192, ldd: 8192, alpha: 1, beta: 1, iters: 1,
   ↪  cold_iters: 0, norm_check: 1}
6
```

This can be executed in rocblas-bench as follows:

```
./rocblas-bench --yaml problem-sizes.yaml
```

The Tensile components (logic files) which rocBLAS uses as the default library are located in the rocBLAS repo under rocBLAS/library/src/blas3/Tensile/Logic. The various sub directories contain the logic files used to build the Tensile library. The most important one is asm_full, which contains the logic used to build the default version of the Tensile library which gets installed with rocBLAS. These directories contain the logic files which are the final output of the tuning as described above. You can examine the content of this directory to see the scope of the full Tensile library product.

4.2 Merging Results into rocBLAS

The process for tuning is done incrementally. We do not perform the tuning on the full set of sizes for each exercise, but rather in stages as the need arises, such as when there is a particular size that is of interest. This means that the final product contains the sizes and kernels from many separate tuning exercises. Therefore, the results of each tuning exercise must be merged. To do so, there is a utility that merges the results of the logic contained in separate files. This script is located in the Tensile repo under Tensile/Tensile/Utilities.

```
usage: merge.py [-h] [--force_merge FORCE_MERGE]
original_dir incremental_dir output_dir

positional arguments:
original_dir          The library logic directory without tuned sizes
```

```
incremental_dir      The incremental logic directory
output_dir           The output logic directory


optional arguments:
-h, --help           show this help message and exit
--force_merge FORCE_MERGE
Merge previously known sizes unconditionally. Default
behavior if not arcturus
```

The merge script takes an input path that contains the previous logic and an incremental directory containing the new logic which is being merged into the previous results—this is the newly tuned logic in build*/3_LibraryLogic. The output path, then, contains the merged results. If you do not want to update the sizes with the lower performance from the new library use --force_merge False flag. We stage everything by merging the results into the asm_full path:

```
ROCBLAS_TENSILE_LOGIC=rocBLAS/library/src/blas3/Tensile/Logic
TENSILE_UTILITIES=Tensile/Tensile/Utilities

# merged the new results into the asm_full
python3 ${TENSILE_UTILITIES}/merge.py ${ROCBLAS_TENSILE_LOGIC}/asm_full newLogicPath
    mergedPath

%# update the ci (continuous integration) path with the update massaged results
%cp mergedPath/* ${ROCBLAS_TENSILE_LOGIC}/asm_ci

# update the full path with the updated results
cp mergedPath/* ${ROCBLAS_TENSILE_LOGIC}/asm_full
```

Once the new results are merged into the rocBLAS logic path, you can rebuild a version of rocBLAS that contains the new logic.

```
./install.sh -c (-d) -a <architecture>  # the -d option installs the dependencies, -
    c build the client apps
```

5 The Tensile Library

The main purpose of Tensile is to generate and organize libraries of kernels to run on AMD GPU architectures. The kernels themselves are basic operation building blocks that form the foundation for solving more complex problems and are common to many different applications. For example, kernels that implement the Generalized Matrix-Matrix Multiplication (GEMM) could ultimately end up being used in more complex machine learning or image processing techniques. The final kernel libraries and master solution library can be loaded by a client using the Tensile API. This API maps user-defined problems to specific kernels that are optimally suited to solve the problem.

TensileCreateLibrary contains components that are suitable for processing solution metadata to generate kernel sources, assemble/compile them, and link them into code object libraries. It is also suitable for combining and managing solution meta-data and their problem mappings. TensileCreateLibrary is mainly used to aggregate a given set of solutions into a master solution library and to generate and bundle associated kernels into code object libraries. The outputs are specifically the master solution library (TensileLibrary.dat) and the code object libraries (.co/.hsaco).

5.1 Master Solution Library

With an aggregate of solutions and their kernels, we need a method for selecting solutions and executing kernels at run time. To see how this works, we need to know a little bit more about the structure of the Master Solution Library. It is inefficient to check *every* solution for the suitability, so instead, they are organized in a hierarchical structure to drastically reduce search time. This process is subject to change, but as of the time of writing, it fits the structure below.

5.2 Library Hierarchy

Libraries are used to implement hierarchical searches. At each level of the hierarchy, the predicates must be asserted before moving to the next level.

Hardware Layer This top level of the hierarchy requires the fewest comparisons as there is a limited amount of supported hardware available. At run time, we can only run kernels built for the hardware installed on the host machine. These are classified by 'gfx' architecture values. See here for supported architectures.

Problem Map The problem map layer uses coarse problem operation classifications. This includes ops like GEMM TT or GEMM TN, which are coded similar to the following: Contraction_l_Alik_Bjlk_Cijk_Dijk. There are a few more comparisons at this level.

Problem The problem library layer is more targeted around specific problem properties. This includes input types or other properties like high precision accumulate. For example, each GEMM kernel can only target specific input and output types and memory mappings; otherwise, it would need to change its implementation drastically.

Size and Speed Finally, problem sizes are matched based on minimum Euclidean distance. Benchmarking is not done for every size possible, so we must match problems to the closest size. Solutions must also pass a final predicate that compares finer details of the problem description (example: CDStridesEqual: true AND KernelLanguageCompatible=ASM, ...). If the predicate fails, then it is not included in the final selection.

At this point, we may have a small pool of kernels that can correctly solve the problem and have performance data for solving a problem of a similar size. Based on what we know of the benchmarking, the kernel with the highest speed is selected to ultimately solve the problem.

IRL Shown below is a real example of a TensileLibrary.yaml file:

```yaml
---
library:
  rows:
  - library:
      map:
        Contraction_l_Alik_Bjlk_Cijk_Dijk:           <-- Problem Operation
          rows:
          - library:
              distance: Euclidean                    <-- Distance measure by
↪   Euclidean method
              properties:
              - {index: 0, type: FreeSizeA}          <-- Size properties used to
↪   measure distance
              - {index: 0, type: FreeSizeB}
              - {index: 0, type: BoundSize}
              table:
              - key: [1, 1, 1]                        <-- Benchmarked size
                speed: 0.00013586956220247663        <-- Benchmarked speed
                value: {index: 53, type: Single}      <-- Solution is index 53!

                ... thousands more benchmark key-pairs

              type: Matching                          <-- Matching table ^^^
            predicate:
              type: And
              value:
              - type: TypesEqual
                value: [Half, Half, Float, Float]
              - {type: HighPrecisionAccumulate, value: true}
            type: Problem                             <-- Problem Layer
        property: {type: OperationIdentifier}         <-- Problem Map Layer
      type: ProblemMap
    predicate:
      type: AMDGPU                                    <-- Hardware Layer
```

```yaml
33        value: {type: Processor, value: gfx900}
34     type: Hardware
35    solutions:                                      <-- Begins the master
   ↪    solution list vvv
36    - debugKernel: false
37      hardwarePredicate: {type: TruePred}
38      ideals: {}
39      index: 0                                       <-- Solution Index
40      info: {1LDSBuffer: '0', AggressivePerfMode: '1', ... lots more
41      }
42      name: Cijk_Alik_Bjlk_HSBH_MT16x16x8_SE_AF0EM2_AMAS3_ASEM2_EPS1_GRVW2_ISA900_K1_
   ↪    KLA_LRVW2_PGR1_PLR1_TT2_2_VW2_WG8_8_1_WGM1
43      problemPredicate:                             <-- Predicate: types must
   ↪    match + HPA + others
44        type: And
45      value:
46      - {index: -1, type: BoundSizeMultiple, value: 2}
47      - {index: 0, type: FreeSizeAMultiple, value: 2}
48      - {type: OperationIdentifierEqual, value: Contraction_l_Alik_Bjlk_Cijk_Dijk}
49      - type: TypesEqual
50      value: [Half, Half, Float, Float]
51      - {type: HighPrecisionAccumulate, value: true}
52      ...
53    problemType: {aType: Half, bType: Half, cType: Float, dType: Float,
   ↪    highPrecisionAccumulate: true,  <-- Problem that kernel can solve
54      operationIdentifier: Contraction_l_Alik_Bjlk_Cijk_Dijk, useBeta: true,
   ↪    useInitialStridesAB: false,
55      useInitialStridesCD: false}
56    sizeMapping:                                    <-- Begin kernel
   ↪    properties meta-data
57      depthU: 8
58      globalAccumulation: false
59      globalSplitU: 1
60      macroTile: [16, 16, 1]
61
62    ... Many more properties
63
64    - debugKernel: false                            <-- Next solution, and
   ↪    so on.
65      hardwarePredicate: {type: TruePred}
66      ideals: {}
67      index: 1
68
69
70
71    ... Many more, hundreds of solutions
72
73
74    ...EOF
```

Due to the yaml indenting and structure, it is a bit difficult to understand the hierarchy just from looking at the file itself; this is why the real example wasn't covered until now. This file is, however, parsed and organized in memory to implement the hierarchy structure previously discussed.

5.3 Code Object Libraries

Kernels are compiled into shared object libraries. These are identified by their file extensions .co/.hsaco. The only difference between them is that .co libs are ASM kernels and .hsaco libs are SOURCE kernels: same ABI, same format.

ASM Kernels   Solutions that have the KernelLanguage: Assembly property use the KernelWriterAssembly object to generate the actual kernel code. This object has the complicated task of translating all of the solution properties, or meta-data, into a sequence of code modules that are eventually rendered into a string of targeted asm code and saved to file as assembly. The ISA property determines the target graphics architecture of the assembly which is important for the object to choose the correct assembly instruction set. For example, the KernelWriterAssembly may dynamically test the assembler for v_mfma instructions for ISA (9,0,6) and will find alternatives if unsupported.

ASM kernels are assembled into .o object files and finally linked into .co files. The file names are obtained from the Solution's Name property.

Source Kernels   Solutions that have the KernelLanguage: Source property uses the KernelWriterSource object to generate the actual kernel code. This object has a similarly complicated task of translating all of the solution properties, or meta-data, into a sequence of code modules that are eventually rendered into a string of C++ code and saved to file as .h and .cpp sources. The ISA property for source kernels is (0,0,0) which means that source kernels are compiled for all architecture targets.

Source kernels are compiled and assembled into .o files and extracted into .hsaco code modules per architecture. The file names are obtained from the Solution's Name property and decorated with the architecture target.

Final Code Object Libraries   Depending on TensileCreateLibrary state for 'MergeFiles', the code object libraries may be linked together into monolithic libraries for each architecture. This only affects the number of library files in the final result of TensileCreateLibrary. Again, the .co and .hsaco files have basically the same format; however, the extensions allow distinguishing between ASM and SOURCE kernels.

5.4 Using the Tensile Library For Testing

The Tensile library's independence from rocBLAS enables us to build and use the libraries as part of a separate workflow. This avoids much of the long wait times related to the rocBLAS build. The rocblas-bench application can reference a library by setting the environment variable ROCBLAS_TENSILE_LIBPATH. When this variable is set to a specific version of the Tensile library, rocBLAS will load that version at runtime; otherwise, the prebuilt version of the library will be loaded. This means that only one version of rocBLAS needs to be built for testing when comparing the results of a new version of the Tensile library with a reference version. When using this method of referencing a library, it is important that the same version of TensileCreateLibrary from the rocBLAS build is used to create the library; otherwise, there could be conflicts.

> ☞ It is important to note here that the rocBLAS project and the Tensile project are independent, so consideration must be given to synchronizing the two projects when building the Tensile library as an independent entity. The logic file that gets generated as a result of Tensile tuning is backward compatible in the sense that they only contain metadata and kernel parameters and not actual code objects. When using Tensile independently of rocBLAS, the two versions of Tensile may not be in sync. TensileCreateLibrary is the utility that generates the code objects and library components which rely on the Tensile client code, and hence, it is important that the generated library is the same one that is used in rocBLAS.

There are two options for ensuring that the version of the library that gets generated uses the same version TensileCreateLibrary as the one used in the rocBLAS build.

Explicit Reference to Tensile During Build   By default rocBLAS, is linked to a specific version (or git commit) of Tensile. You can override the default reference and import a local version of Tensile into rocBLAS during the build using the -t option of install.sh when building rocBLAS. In this case, build rocBLAS using

```
./install.sh -c -t /path/to/repo/Tensile
```

This will ensure that the version of Tensile rocBLAS uses is the one referenced to during the build (/path/to/repo/Tensile in the example).

Explicit Reference to TensileCreateLibrary   The version of TensileCreateLibrary that was imported during the rocBLAS build can be referenced. The is found in the following path:

```
/repo/rocBLAS/build/release/Tensile/Tensile/bin/TensileCreateLibrary.
```

5.4.1   Example Usage

Once the library is generated, it can be used in rocBLAS by setting the environment variable ROCBLAS_TENSILE_LIBPATH. Execute rocblas-bench as follows:

```
ROCBLAS_TENSILE_LIBPATH=path/to/drop/new/tensile/library ./rocblas-bench --yaml
    problem_sizes.yaml
```

5.4.2   The TensileCreateLibrary Command

The TensileCreateLibrary application takes, as input, a path containing the Tensile logic files and generates a Tensile library. The generated library contains the library artifacts and code objects that contain the kernels as well as the selection logic—that is, the logic for selecting the kernel to be executed. An example execution of the create library application is shown below.

```
Tensile/bin/TensileCreateLibrary <Options...> <LogicPath> <OutputPath> <
    RuntimeLanguage>

[-h] Display usage help

[--cxx-compiler {hipcc}] Compiler override (default hipcc).

[--code-object-version {default,V4,V5}] Code object version override.

[--architecture {all,gfx000,gfx803,gfx900,gfx906,gfx908}] Architecture override
    (default all).

[--merge-files] / [--no-merge-files] If merge, all kernels are merged to one code
    object file per architecture (default on).

[--short-file-names] / [--no-short-file-names] Windows only option for shortening
    output files names. Currently disabled.

[--library-print-debug] / [--no-library-print-debug] Solutions will print enqueue
    info when enqueueing a kernel (default off)

[--no-enumerate] Disable enumeration of host graphics architecture.

[--package-library]

[--embed-library EMBEDLIBRARY] Embed (new) library files into static variables.
    Specify the name of the library.

[--embed-library-key EMBEDLIBRARYKEY] Prefix embedded symbols with EMBEDLIBRARYKEY.

[--version VERSION] Embed version into library files.

[--generate-manifest-and-exit] In the output directory, create a manifest file of
    expected outputs only.

[--library-format {yaml,msgpack}] Choose format of output library (default msgpack).
    Respective file extensions {.yaml,.dat}

LogicPath Path to LibraryLogic yaml files.
```

19

```
OutputPath Output directory.
```

```
{OCL,HIP,HSA} Chosen runtime language.
```

To run TensileCreateLibrary use the following command. The input to this script is the library logic generated by Tensile.

```
/path/to/tensile-rep/Tensile/bin/TensileCreateLibrary --merge-files --no-short-file-
    names --no-library-print-debug --code-object-version=default --cxx-compiler=
    hipcc --library-format=msgpack path/to/new/logic path/to/drop/new/tensile HIP
```

6 Tuning Automation

Tensile provides a set of automation scripts in Tensile/tuning/automation for simplifying the tuning process. Starting with a set of logs that contain information about the problems sizes, the automation tools will generate the tuning configurations and then start a first pass at tuning based on the generated configuration. In the current version of automation, the configuration generator is not very smart and only provides a simple classification based on size categories. Hence, the full tuning is unlikely to produce the optimal performance for all sizes under consideration and further tuning iterations will be necessary. Despite this limitation, it does set up a framework and a quick tuning pass as part of the tuning workflow, so it is a good place to start a tuning exercise.

6.1 Automation Workflow
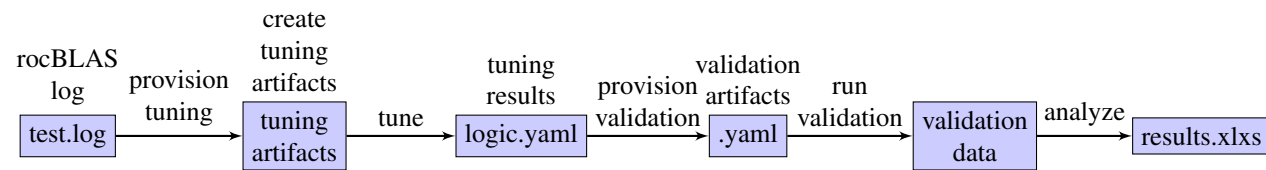
Figure 2: Tensile automation workflow

6.2 rocBLAS Logging Information

rocBLAS logging provides information about the rocBLAS functions which get called during an application as well as the size information of the calls. This logging is enabled by setting the ROCBLAS_LAYER environment variable. The two settings that get used as input into the tuning automation are ROCBLAS_LAYER=2 and ROCBLAS_LAYER=4; these each produce output about the rocBLAS calls in different formats.

```
ROCBLAS_LAYER=2
```

```
# generates output in the form of
rocblas-bench -f gemm -r s --transposeA N --transposeB N -m 1024 -n 1024 -k 1024 --
    beta 0.0 --alpha 1.0 --lda 1024 --ldb 1024 --ldc 1024
```

```
ROCBLAS_LAYER=4
```

```
# generates output in the form of
- { rocblas_function: rocblas_sgemm, transA: N, transB: N, M: 1024, N: 1024, K:
    1024, lda: 1024, ldb: 1024, ldc: 1024}
```

In either case, the automation tools are able to parse out the relevant information about the sizes and use it to generate the tuning artifacts which drive the automation.

6.3 Tuning Automation Scripts

The typical tuning exercise starts with a particular application which links to the rocBLAS library and calls gemm. The application is run with logging enabled by setting the ROCBLAS_LAYER environment variable. The output of the log is used as input to the tuning automation.

All of the tuning scripts described below take a working directory as their first argument. For a given tuning exercise, this should be the same directory for each script as subsequent scripts require the outputs of previous ones.

6.3.1 The provision_tuning.sh Script

The first step of the tuning is set up using the provision_tuning.sh script. This script generates a set of configurations and other artifacts used during tuning, as well as provisions a copy of the Tensile development repo if a local version of Tensile is not specified. An example call is

```
./tuning/scripts/provision_tuning.sh tensile_tuning logs/inception-rocblas-
    configs_unique.log tf_inception.yaml vega20
```

All possible arguments for the script are as follows:

```
Usage: ./provision_tuning.sh WORKING_PATH LOG_PATH OUTPUT_SUFFIX.yaml LIBRARY [
    options]
  where LIBRARY = {arcturus | vega20 | vega10 | mi25 | r9nano | hip}


Options:
-h | --help                 Display this help message
-n | --network NAME         Neural network name. If this is set, LOG_PATH should
    be a directory. Will only tune log files with this string in the file name
-p | --tensile-path PATH    Path to existing Tensile (will not provision new copy)
Options for provisioning Tensile:
-f | --tensile-fork USERNAME  Tensile fork to use
-b | --branch BRANCH        Tensile branch to use
-c | --commit COMMIT_ID     Tensile commit to use
-t | --tag GITHUB_TAG       Tensile tag to use
-i | --id ID                ID to append to Tensile directory name
Options for config generation:
-t | --tile-aware          Use tile-aware method. (limited support)
-m | --mfma                Use MFMA kernels
-r | --rk                  Use replacement kernels (sgemm only)
-s | --disable-strides     Disable leading dimensions and strides in tuning file
--initialization {rand_int | trig_float | hpl} (=rand_int)  Data initialization for
    matrices
--problem-definition {gemm | batch | both} (=both)         Which problem types to
    tune
--client {new | old | both} (=new)                         Which Tensile runtime
    client to use
```

After this script is run, the following artifacts are generated and available in the tensile_tuning directory, which is created as a result of the script.

- The **configs** directory contains the benchmark configuration/input yaml files used by Tensile

- The **make** directory contains scripts for running the generated configs with Tensile

- The **scripts** directory has rocblas-bench scripts and rocblas-bench yaml files. Currently only the yaml files are used

- The **scripts2** directory has rocblas-bench scripts and rocblas-bench yaml files with the ROCBLAS_TENSILE_LIB_CLIENT environment variable set for the tuned Tensile library. Currently unused

- The **sizes** directory has a csv file that shows the parsed log file parameters in a comma-separated format

- If a local version of Tensile is not provided, the **tensile/Tensile** directory containes the provisioned copy of Tensile.

6.3.2 The provision_verification.sh Script

The provision_verification.sh prepares the library files for generating the tuned Tensile library for use with rocBLAS and builds said Tensile library. It also provisions and builds a copy of rocBLAS if a local version is not specified. An example call is

```
./tuning/scripts/provision_verification.sh tensile_tuning tensile_tuning/tensile/
    Tensile vega20
```

All possible arguments for the script are as follows:

```
Usage: ./provision_verification.sh WORKING_PATH TENSILE_PATH LIBRARY [options]
  where LIBRARY = {arcturus | vega20 | vega10 | mi25 | r9nano | hip}

Options:
-h | --help                 Display this help message
-p | --rocblas-path PATH    Path to existing rocBLAS (will not provision new copy)
-f | --rocblas-fork USERNAME  rocBLAS fork to use
-b | --branch BRANCH        rocBLAS branch to use
-c | --commit COMMIT_ID     rocBLAS commit to use
-t | --tag GITHUP_TAG       rocBLAS tag to use
-i | --id ID                ID to append to rocBLAS directory name
-s | --sclk                 Frequency of sclk in MHz
-n | --no-merge             Skip merge step
--no-massage                Skip massage step
--log-dir                   Directory for logs
--redo                      Force logic preparation, merge, massage, and library
    build steps to be redone
```

After running the script, the **logs** directory will contain the logs for the merge, massage, and rocBLAS build steps. The **library** directory will have the exact, massage, and merge library logic files available:

- **exact** contains the library file(s) available in 3_LibraryLogic after running Tensile using the config file(s)

- **merge** has the exact file(s) merged with the file(s) that are already in rocBLAS

- **massage** adds the special Ldc/Ldd kernels to the merged files (only if using gfx900/906 and hpa hgemm sizes are not used)

If a local version of rocBLAS is not provided, the **rocblas/rocBLAS** directory contains the provisioned (and built) copy of rocBLAS. Note that this script should be run with `--no-massage` if tuning for arcturus or if tuning any hpa-hgemm sizes.

6.3.3 The run_validation.sh Script

The run_validation.sh script benchmarks the problems for both the reference and tuned versions of the Tensile library. An example call is

```
./tuning/scripts/run_validation.sh tensile_tuning tensile_tuning/rocBLAS/
```

After running the script, the **benchmarks** directory will contain the benchmark results for the reference and tuned versions of the library in the respective subdirectories.

6.3.4 The analyze_results.sh Script

Finally, after the results are benchmarked using the reference and tuned versions of Tensile, they may be analyzed using the analyze_results.sh script. An example call is

```
./tuning/scripts/analyze_results.sh tensile_tuning inception-rocblas-configs_unique.
    log vega20 -s 2 -f 1301
```

All possible arguments for the script are as follows:

```
Usage: ./analyze_results.sh WORKING_PATH LOG_PATH LIBRARY [options]
  where LIBRARY = {arcturus | vega20 | vega10 | mi25 | r9nano | hip}

Options:
-h | --help            Display this help message
-s | --size            Data size
-f | --sclk            Frequency of sclk in MHz
-m | --mfma            Was MFMA enabled
-c | --count           Sets all cases where count=1 to count=10
-n | --no-plot         Skip plotting
```

After running this script, the **analysis** directory will contain analysis of the reference and tuned benchmark results along with a comparison of the two—all in respective subdirectories.

6.3.5 Automation Usage Example

```
# provisions Tensile for tuning
./tuning/scripts/provision_tuning.sh tensile_tuning logs/inception-rocblas-
    configs_unique.log tf_inception.yaml vega20

cd tensile_tuning/make

# does the actual tuning using the generated configuration
./runTensileTuning-all.sh

cd ../..

# provisions the verifications, including building rocblas-bench
./tuning/scripts/provision_verification.sh tensile_tuning tensile_tuning/tensile/
    Tensile vega20

# runs the actual benchmarks for the old and new versions of the library
./tuning/scripts/run_validation.sh tensile_tuning tensile_tuning/rocBLAS/

# analysis of the results
./tuning/scripts/analyze_results.sh tensile_tuning inception-rocblas-configs_unique.
    log vega20 -s 2 -f 1301
```

6.3.6 The master_tuning_script.sh Script

Each of the previous scripts can be run independently, separating out each of the steps in one pass of the tuning workflow. The master_tuning_script.sh script combines the complete workflow without intervention. It takes the same rocBLAS log file information as described in the previous part of this section as input and runs the full process end to end. All possible arguments for the script are as follows:

```
Usage: ./master_tuning_script.sh WORKING_DIR LOG_PATH [options]

Options:
-h | --help            Display this help message
-n | --network         String to search for in filenames in log directory
--tensile-path PATH    Path to existing Tensile (will not provision new copy)
--rocblas-path PATH    Path to existing rocBLAS (will not provision new copy)
-y | --data-type       Data type of sizes that you want to tune (sgemm, dgemm,
    hgemm only)
-t | --tile-aware      Use tile-aware method. (limited support)
-m | --mfma            Use MFMA kernels
-r | --rk              Use replacement kernels (sgemm only)
-s | --disable-strides Disable leading dimensions and strides in tuning file
```

```
-f | --sclk             Frequency of sclk in MHz
-c | --count            Sets all cases where count=1 to count=10
-d | --dependencies     Install required dependencies (dependencies are not
   installed by default)
--redo                  Force logic preparation, merge, massage, and library build
   steps to be redone
-l | --library \
   {arcturus | vega20 | vega10 | mi25 | r9nano | hip}        GPU used for tuning (
      arcturus, mi25, mi50, mi60, r7, v340 only)
--initialization {rand_int | trig_float | hpl} (=rand_int)  Data initialization for
   matrices
--problem-definition {gemm | batch | both} (=both)          Which problem types to
   tune
--client {new | old | both} (=new)                          Which Tensile runtime
   client to use
```

7   Tuning Considerations

- Kernels can be compute-bound or memory-bound?

  - If a kernel is memory bound, throttling will occur, and performance will drop?

- To avoid throttling, set the system clock (sclk) and memory clock (memclk) to a near one-to-one ratio?

  - Can be done using rocm-smi or atitool?

- CPU timers are used by rocblas-bench, while GPU timers are used by Tensile?

  - Generally leads to rocblas-bench producing lower performance than Tensile?

- Timing is affected by launching rocblas-bench?

  - Leads to varying performance?

- Reliance on GFlops instead of efficiency. The GFlops may be affected by environmental factors, so the results on one machine may not be the same as those on another. For example, the clock rate that was used during the tuning process may differ between systems or even on the same machine at different times. To compensate for this, it is recommended that the efficiency is use as a performance comparison when comparing results across different systems and tuning exercises. Tensile/tuning/ConvertToEfficiency.py converts the GFlops in the library logics to efficiency.

$$\text{efficiency} = \frac{\text{kernel performance} \times 1000}{o \times f \times \text{numCUs}} \times 100$$

  - f = sclk frequency in MHz
  - numCUs = number of compute units
  - o = ops/CU/cycle
  - 1000 = conversion to MFlops/sec from GFlops/sec

A   ROCm and Tensile Installation and References

Use the following references to find the documentation about the ROCm platform.

- Documentation for the AMD ROCm platform.

- Reference to rocBLAS.

- Reference to Tensile provisioning overview and Tensile wiki.

A.1 ROCm Installation

Follow these links to find information about installing the ROCm platform.

- [Install ROCm dependancies](#).

- rocBLAS [building and install](#).

A.2 Install Tensile Dependencies

Use the following steps to install the Tensile and automation dependencies.

Install the python dependencies:

```
sudo apt install -y --no-install-recommends cmake make ca-certificates git \
pkg-config python3 python3-dev python3-matplotlib python3-pandas python3-pip \
python3-setuptools python3-tk python3-venv python3-yaml libnuma1 llvm-6.0-dev \
libboost-all-dev zlib1g-dev libomp-dev gfortran libpthread-stubs0-dev \
libmsgpack-dev libmsgpackc2 wget
```

Then install:

```
pip3 install setuptools --upgrade && pip3 install wheel && pip3 install pyyaml
    msgpack openpyxl pandas
```

Be cautious about the cmake version. If you get an error about upgrading your cmake version, make sure you install a version that is compatible with the Tensile version.

B Tensile Kernel Naming Convention

When Tensile or Tensile Create Library produces the kernels, it generates names for those kernels based on the parameters which are used to generate each kernel. The naming convention used is based on the name of the parameters involved. The part of the name produced from the parameter is an abbreviation of the name concatenated with the value of the parameter. For example, GlobalSplitU = 4 will produce a name part GSU4. The full name of a kernel looks like the following:

```
Cijk_Ailk_Bljk_SB_MT128x128x16_SE_GSU1_K1_TT8_8_WG16_16_1.s.
```

Any library will contain a large set of parameters that are common to all the kernels in the library so, in the final library, those parameters are not included as part of the naming of the kernels in that library. This reduces the name as the full set of parameters is rather large.

C Parameter Name Abbreviation Mapping

Table 1: Parameter names and their abbreviations. Most abbreviations are a combination of the first letters of each word in a parameter's name. Consequently, abbreviations are not guaranteed to be unique. Derived parameters are not listed.

Parameter Name	Abbreviation
1LDSBuffer	1LDSB
AggressivePerfMode	APM
AssertAlphaValue	AAV
AssertBetaValue	ABV
AssertCEqualsD	ACED
AssertFree0ElementMultiple	AF0EM
AssertFree1ElementMultiple	AF1EM
AssertMinApproxSize	AMAS
AssertSizeEqual	ASE
AssertSizeGreaterThan	ASGT
AssertSizeLessThan	ASLT
AssertSizeMultiple	ASM

AssertStrideAEqual	ASAE
AssertStrideBEqual	ASBE
AssertStrideCEqual	ASCE
AssertStrideDEqual	ASDE
AssertSummationElementMultiple	ASEM
AtomicAddC	AAC
BufferLoad	BL
BufferStore	BS
CheckDimOverflow	CDO
CheckTensorDimAsserts	CTDA
CustomKernelName	CKN
DepthU	DU
DepthULdsDivisor	DULD
DirectToLds	DTL
DirectToVgprA	DTVA
DirectToVgprB	DTVB
DisableAtomicFail	DAF
DisableKernelPieces	DKP
DisableVgprOverlapping	DVO
EdgeType	ET
ExpandPointerSwap	EPS
Fp16AltImpl	F16AI
FractionalLoad	FL
GlobalRead2A	GR2A
GlobalRead2B	GR2B
GlobalReadCoalesceGroupA	GRCGA
GlobalReadCoalesceGroupB	GRCGB
GlobalReadCoalesceVectorA	GRCVA
GlobalReadCoalesceVectorB	GRCVB
GlobalReadPerMfma	GRPM
GlobalReadVectorWidth	GRVW
GlobalSplitU	GSU
GlobalSplitUAlgorithm	GSUA
GlobalSplitUSummationAssignmentRoundRobin	GSUSARR
GlobalSplitUWorkGroupMappingRoundRobin	GSUWGMRR
GroupLoadStore	GLS
InnerUnroll	IU
InterleaveAlpha	IA
ISA	ISA
KernelLanguage	KL
LdcEqualsLdd	LEL
LdsBlockSizePerPad	LBSPP
LdsPadA	LPA
LdsPadB	LPB
LocalDotLayout	LDL
LocalRead2A	LR2A
LocalRead2B	LR2B
LocalReadVectorWidth	LRVW
LocalWrite2A	LW2A
LocalWrite2B	LW2B
LocalWritePerMfma	LWPM
LoopDoWhile	LDW
LoopTail	LT
MACInstruction	MAC
MacroTile	MT
MacroTileShapeMax	MTSM
MacroTileShapeMin	MTSM
MagicDivAlg	MDA
MatrixInstruction	MI
MaxOccupancy	MO
MaxVgprNumber	MVN
MIArchVgpr	MIAV

MinVgprNumber	MVN
NonTemporalA	NTA
NonTemporalB	NTB
NonTemporalC	NTC
NonTemporalD	NTD
NoReject	NR
NumElementsPerBatchStore	NEPBS
NumLoadsCoalescedA	NLCA
NumLoadsCoalescedB	NLCB
OptNoLoadLoop	ONLL
OptPreLoopVmcnt	OPLV
PackBatchDims	PBD
PackFreeDims	PFD
PackGranularity	PG
PackSummationDims	PSD
PerformanceSyncLocation	PSL
PerformanceWaitCount	PWC
PerformanceWaitLocation	PWL
PersistentKernel	PK
PersistentKernelAlongBatch	PKAB
PrefetchAcrossPersistent	PAP
PrefetchAcrossPersistentMode	PAPM
PrefetchGlobalRead	PGR
PrefetchLocalRead	PLR
ReplacementKernel	RK
ScheduleGlobalRead	SGR
ScheduleIterAlg	SIA
ScheduleLocalWrite	SLW
SourceSwap	SS
StaggerU	SU
StaggerUMapping	SUM
StaggerUStride	SUS
StoreCInUnroll	SCIU
StoreCInUnrollExact	SCIUE
StoreCInUnrollInterval	SCIUI
StoreCInUnrollPostLoop	SCIUPL
StorePriorityOpt	SPO
StoreRemapVectorWidth	SRVW
StoreSyncOpt	SSO
StoreVectorWidth	SVW
SuppressNoLoadLoop	SNLL
ThreadTile	TT
TransposeLDS	TLDS
UnrollIncIsDepthU	UIIDU
UnrollMemFence	UMF
Use64bShadowLimit	U64SL
UseInstOffsetForGRO	UIOFGRO
UseSgprForGRO	USFGRO
VectorAtomicWidth	VAW
VectorStore	VS
VectorWidth	VW
WavefrontSize	WS
WaveSeparateGlobalReadA	WSGRA
WaveSeparateGlobalReadB	WSGRB
WorkGroup	WG
WorkGroupMapping	WGM
WorkGroupMappingType	WGMT

D Library Logic Specifications

The following is a full list of platform targets that can be used for the library logic generation in Tensile.

```
1
2  ScheduleName: "vega20"
3  DeviceNames: ["Device 66a0", "Device 66a1", "Device 66a7", "Device 66af", "Vega
   ↪   20"]
4  ArchitectureName: "gfx906"
5
6  ScheduleName: "vega10"
7  DeviceNames: ["Device 6863", "Device 6862", "Device 687f", "Device 6860", "Device
   ↪   6861", "Vega 10 XTX [Radeon Vega Frontier Edition]", "Vega [Radeon RX Vega]",
8  "Vega 10 XT [Radeon RX Vega 64]", "Vega", "Device 6864", "Device 686c", "Vega 10
   ↪   [Radeon Instinct MI25 MxGPU]"]
9  ArchitectureName: "gfx900"
10
11 ScheduleName: "mi25"
12 DeviceNames: ["Device 6860"]
13 ArchitectureName: "gfx900"
14
15 ScheduleName: "r9nano"
16 DeviceNames: ["Device 7300"]
17 ArchitectureName: "gfx803"
18
19 ScheduleName: "hip"
20 DeviceNames: ["Device 0000"]
21 ArchitectureName: "fallback"
22
```