# Benchmark Suites: Developing and Implementing a Simple Benchmark Suite

By\ Khalid Al-Hawaj

COE403/COE501 Term 242

## 1 Introduction

Computer architecture is the practice of designing, reasoning, implementing, testing, and tuning computer systems through structural and hierarchical methods targeting different abstraction layers in the compute stack. The current standard design of the different architectures and micro-architectures have been the product of historical iterative process of benchmarking and innovation; in other words, well-designed benchmark applications are executed on the processor. The execution of the benchmark is further studied to understand the execution advantages and disadvantages; finally, innovation is proposed and motivated to increase performance by further leveraging the advantages and eliminating the disadvantages. Through benchmarking, computer architects are able to test and evaluate different aspects of the architecture and micro-architecture designs; subsequently, computer architects address any shortcoming in the architecture and micro-architecture to improve performance, thus yielding innovation.

To enable effective benchmarking, the common practice is to leverage benchmark suites. A benchmark suite is composed of: (1) a flexible and extensible build system; (2) multiple functions and methods to facilitate effective benchmarking; and (3) multiple independent benchmarks. The *build system* will take as an input all defined benchmarks and compile them into independent binaries to be executed and profiled. To control how the benchmarks are executed and to help profile these benchmarks on different architectures and operating systems, the benchmark suites provides useful *functions and methods*. These functions and methods can help schedule the benchmark execution on the machine; moreover, some functions can be provided to facilitate specialized mathematical computations optimized for the targeted architectures. Most importantly, some functions and methods are provided to help with profiling the benchmarks. Profiling a benchmark entails characterizing its runtime, cache misses, TLB misses, page faults, and many more. The crown jewels of a benchmark suite are the benchmarks themselves. By leveraging the build system, benchmarks are compiled into independent binaries. With the use of the available libraries (e.g., the benchmark suite's functions and methods), the benchmark binaries are executed and profiled to further study and compare different architectural details.

An effective benchmark suites offer extensible build systems that can be expanded to accommodate new benchmarks. To enable studying different architectures, a build system of a benchmark suite would offer the notion of *implementations* (abbreviated as `impls`) of benchmarks. An *implementation* (abbreviated as `impl`) of the benchmark is a code that achieves the functionality required by the benchmark (i.e., performs the same task); however, each *implementation* is tailored to leverage a specific architecture. Some `impls` *could* be executed on a different architecture than the one for which it was optimized, but it might perform worse; it is, also, possible that some `impls` would not be able to execute on any other architecture than the one which these `impls` target.

In this assignment, you are asked to implement a new benchmark and add it to the provided educational benchmark suite. You are tasked with understanding the build system of this benchmark suite. Also, you are tasked with understanding the algorithm and all implementation-details of the new benchmark. Finally, you are tasked with running and profiling the benchmark on your machine to see the benefits of specialized implementations.

```
CC=gcc
LDFLAGS=-lm
CFLAGS=-g -O3

my_first_prog: my_first_prog.o
    $(CC) $(LDFLAGS) my_first_prog.o -o my_first_prog

my_first_prog.o: my_first_prog.c
    $(CC) -I. $(CFLAGS) -c my_first_prog.c -o my_first_prog.o

clean:
    rm -f my_first_prog my_first_prog.o

.PHONY: clean

```

Figure 1: **A Simple Makefile To Compile and Link `my_first_prog`.**

The remaining sections are as follows: Section 2 explains the structure of simple benchmark suite; Section 3 discusses the required kernel to be implemented as a benchmark; Section 4 details the deliverables and the structure of the report to be submitted.

## 2 YABMS: Yet Another Benchmark Suite

To understand the purpose of benchmark suites and appreciate their contribution in designing high-performance computer architectures, in this assignment, you are asked to develop and integrate new benchmarks into YABMS[3]—a simple and educational benchmark suite. YABMS is composed of three essential parts: (1) simple build system; (2) macros and functions to facilitate running the benchmarks; and (3) benchmarks implementing kernels and full applications with real-world relevance.

This section will discuss the different parts of YABMS. Subsection 2.1 details the simple build system. Subsection 2.2 documents the different macros and functions provided by YABMS. Subsection 2.3 goes over the `vvadd` benchmark as an example.

### 2.1 YABMS Build System

To enable extensible building and linking of different benchmarks into independent binaries, YABMS employs a simple build system. For the most part, the build system is fully automated; however, for certain situations where a customized build routine is required, the build system can be easily modified to accommodate. At the core, YABMS' build system was developed using Makefiles[1], which conforms to the Section 6.2 of IEEE Standard 1003.2-1992[2]. A Makefile defines rules to build targets; the targets of the rules are files or directories to be created or updated. Each rule has a list of dependencies and a recipe. The dependencies are files, which in-turn could be a defined target of other rules. The recipe is indented by a tab and composed of multiple lines. Each line in the recipe is a shell command to be executed. If any shell command results in a non-zero exit code, the recipe fails and stops. To determine which targets require updating, GNU `make` keeps track of the `last-modified` timestamp of targets and their dependencies. A target requires updating (i.e., re-triggering the rule) if the `last-modified` timestamp of one of its dependencies is later than the target's own `last-modified` timestamp.

Figure 1 shows a very simple Makefile to build a program called "`my_first_prog`". The first defined rule will be the default target. If "`make`" is executed with no specified target, the default target will be chosen. The Makefile defines a rule for building the target "`my_first_prog`", which has one dependency (i.e., the object file `my_first_prog.o`) and a recipe. To check whether `my_first_prog` is up-to-date, the "`make`" checks on its dependencies. The only dependency, `my_first_prog.o`, is defined as a target for a different rule; as a result, "`make`" checks on its dependencies. There is one dependency for `my_first_prog.o`, which is `my_first_prog.c`. Since `my_first_prog.c` is not defined as a target for any rule, "`make`" does not know how to build it, so "`make`" stops there. Then, "`make`" checks the `last-modified` timestamp of `my_first_prog.c` and make sure it is earlier than `my_first_prog.o`. If that is the case, then `my_first_prog.o` is up-to-date and nothing else to do. If not, "`make`" re-triggers the rule and executes the commands defined in the recipe. Then, "`make`" trace back to `my_first_prog`

```
CC=gcc
LDFLAGS=-lm
CFLAGS=-g -O3

my_first_prog: my_first_prog.o
	$(CC) $(LDFLAGS) my_first_prog.o -o my_first_prog

%.o: %.c
	$(CC) -I. $(CFLAGS) -c $< -o $@

clean:
	rm -f my_first_prog my_first_prog.o

.PHONY: clean
```

Figure 2: **A Makefile To Compile and Link `my_first_prog` Using Pattern Rules.**

and checks if its `last-modified` timestamp is later than its dependencies; if not, then "`make`", again, re-triggers its rule and executes the commands in the recipe.

Figure 2 shows a Makefile that exemplifies the use of pattern rules. A pattern rule uses `%` to match any non-empty string. One can think of the `%` as a wildcard character. When inspecting the definition of the pattern rule in Figure 2, one can notice the use of automatic variables. Automatic variables are computed and evaluated automatically for each rule to aid and assist with its execution. There are many automatic variables as defined by GNU Makefile [1]. The pattern rule in the Makefile shown in Figure 2 uses two automatic variables: "`$<`" and "`$@`". The first automatic variable, "`$<`", holds the target as a string; while the second automatic variable, "`$@`", holds all the dependencies as a string. Since the rule uses the "`%`" wildcard character, the automatic variables can help writing the recipe that can specify the target and its dependencies dynamically instead of hardcoding filenames and directories.

If a target does not have a recipe and does not exist as a file or a directory, "`make`" throws an error. Since every target is treated as a file or directory, we can define `phony` targets. These targets are defined as normal, however, we tell "`make`" that these targets do not represent actual files or directories that would be created. Since these phony targets are not associated with any file or directory, "`make`" always re-trigger any recipe for a target that has them as dependencies.

## 2.2 YABMS Functions and Macros

The YABMS benchmark suite provides multiple useful functions and libraries that can aid with testing and facilitating accurate performance profiling. Currently, only runtime is being profiled. Figure 3 shows macros provided to facilitate collecting runtime statistics about the execution of a benchmark. YABMS provides macros to facilitate data allocation and initialization, shown in Figure 4. Additionally, YABMS provides macros that provide testing and comparing the produced results, shown in Figure 5.

## 2.3 YABMS Benchmarks: `vvadd`

As an example, YABMS provides `vvadd`–a complete benchmark implemented with its different `impls`. You can inspect the code and understand fully how to leverage the different provided macros and functions. The benchmarks are defined as application with multiple implementations "`impl`". The Makefile automatically picks up any directory with the file "`Makefile.mk`". The following lines in the "`Makefile.mk`" inside the "`src`" directory searches for all directories with "`Makefile.mk`", and then includes these Makefile fragments, which in-turn imports all rules from these "`Makefile.mk`" files:

```
# Directory
_LOCAL_DIR := $(shell dirname $(realpath $(lastword $(MAKEFILE_LIST))))

# Find all applications
_MK_FILES := $(shell find $(_LOCAL_DIR) -mindepth 2 -maxdepth 2 -name "Makefile.mk")

-include $(_MK_FILES)
```

```c
#define __DECLARE_STATS(_num_runs, _num_stdev)          \
  /* Time keeping */                                    \
  struct timespec ts;                                   \
  struct timespec te;                                   \
                                                        \
  /* Iterate and average runtimes */                    \
  uint32_t num_runs = _num_runs;                        \
  uint64_t* runtimes;                                   \
  bool* runtimes_mask;                                  \
                                                        \
  runtimes = (uint64_t*)calloc(num_runs,                \
                               sizeof(uint64_t));       \
                                                        \
  runtimes_mask = (bool*)calloc(num_runs,               \
                                   sizeof(bool));       \
                                                        \
  /* Constants for statistical analysis */              \
  const unsigned int nstd = _num_stdev;

#define __DESTROY_STATS()                               \
  free(runtimes);                                       \
  free(runtimes_mask);

#define __SET_START_TIME() {                            \
  __COMPILER_FENCE_;                                    \
  if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1) {      \
    printf("\n\n    ERROR: getting time failed!\n\n"); \
    exit(-1);                                           \
  }                                                     \
}

#define __SET_END_TIME() {                              \
  __COMPILER_FENCE_;                                    \
  if (clock_gettime(CLOCK_MONOTONIC, &te) == -1) {      \
    printf("\n\n    ERROR: getting time failed!\n\n"); \
    exit(-1);                                           \
  }                                                     \
}

#define __CALC_RUNTIME() ({                             \
    (((te.tv_sec  - ts.tv_sec ) * 1e9) +                \
      (te.tv_nsec - ts.tv_nsec)      ) ;                \
})

```

Figure 3: **Provided macros to perform runtime profiling.**

```c
#define __ALLOC_DATA(type, nelems) ({                      \
  unsigned int nbytes = (nelems) * sizeof(type);        \
  type* temp = (type*)aligned_alloc(512 / 8, nbytes);  \
                                                          \
  if (temp == NULL) {                                     \
    printf("\n");                                         \
    printf("  ERROR: Cannot allocate memory!");          \
    printf("\n");                                         \
    printf("\n");                                         \
    exit(-2);                                             \
  }                                                       \
                                                          \
  temp;                                                   \
})

#define __ALLOC_INIT_DATA(type, nelems) ({                 \
  unsigned int nbytes = (nelems) * sizeof(type);        \
  type* temp = (type*)aligned_alloc(512 / 8, nbytes);  \
                                                          \
  if (temp == NULL) {                                     \
    printf("\n");                                         \
    printf("  ERROR: Cannot allocate memory!");          \
    printf("\n");                                         \
    printf("\n");                                         \
    exit(-2);                                             \
  }                                                       \
                                                          \
  /* Generate data */                                     \
  for(int i = 0; i < nelems; i++) {                       \
    temp[i] = rand() % (0x1llu << (sizeof(type) * 8)); \
  }                                                       \
  temp;                                                   \
})

```

Figure 4: **Provided macros to perform data allocation and initialization.**

```c
#define __SET_GUARD(array, sz) {                                \
  ((byte*)array)[sz + 0] = 0xfe;                                \
  ((byte*)array)[sz + 1] = 0xca;                                \
  ((byte*)array)[sz + 2] = 0xad;                                \
  ((byte*)array)[sz + 3] = 0xde;                                \
}

#define __CHECK_MATCH(ref, array, sz) ({                        \
  bool __tmp = true;                                            \
                                                                \
  for(int i = 0; (i < sz) && __tmp; i++) {                      \
    __tmp = __tmp && (ref[i] == array[i]);                      \
  }                                                             \
                                                                \
  __tmp;                                                        \
})

#define __CHECK_FLOAT_MATCH(ref, array, sz, delta) ({           \
  bool __tmp = true;                                            \
                                                                \
  for(int i = 0; (i < sz) && __tmp; i++) {                      \
    __tmp = __tmp && (fabs(ref[i] - array[i]) < delta);\
  }                                                             \
                                                                \
  __tmp;                                                        \
})

#define __CHECK_GUARD(array, sz) ({                             \
  bool match = true;                                            \
                                                                \
  match = match && (((byte*)array)[sz + 0] == 0xfe);           \
  match = match && (((byte*)array)[sz + 1] == 0xca);           \
  match = match && (((byte*)array)[sz + 2] == 0xad);           \
  match = match && (((byte*)array)[sz + 3] == 0xde);           \
                                                                \
  match;                                                        \
})
```

Figure 5: **Provided macros to perform testing.**

```makefile
define template_mk

# Name and directories
$(1)_BIN := $(1)
$(1)_DIR := $(2)
$(1)_BUILD_DIR := $$(BUILD_DIR)/$(1)_build

# Object files
$(1)_C_FILES := $$(shell find $$($(1)_DIR) -name "*.c")
$(1)_O_FILES := $$(foreach x,$$($(1)_C_FILES),$$(patsubst $$($(1)_DIR)/%,$$($
    (1)_BUILD_DIR)/%,$$(x)))
$(1)_O_FILES := $$(foreach x,$$($(1)_O_FILES),$$(patsubst %.c,%.o,$$(x)))
$(1)_D_FILES := $$($(1)_O_FILES:%.o=%.d)

# Include directories
$(1)_INCLUDE_DIR := $$($(1)_DIR) $$(SRC_DIR) $$(BUILD_DIR) $$($(1)_BUILD_DIR)

# Generate include directory argument
$(1)_INCLUDES := $$(foreach x,$$($(1)_INCLUDE_DIR),$$(addprefix -I,$$(x)))

# GCC -MMD glory!
-include $$($(1)_D_FILES)

$$($(1)_BUILD_DIR)/%.o: $$($(1)_DIR)/%.c | $$($(1)_BUILD_DIR)
	mkdir -p $$(dir $$@)
	$$(CC) $$($(1)_INCLUDES) $$(CFLAGS) -MMD -c $$< -o $$@

$$(BUILD_DIR)/$$($(1)_BIN): $$($(1)_O_FILES) | $$($(1)_BUILD_DIR)
	$$(CC) $$($(1)_O_FILES) $$(IFLAGS) -o $$@

$$($(1)_BUILD_DIR): | $$(BUILD_DIR)
	mkdir -p $$@

$(1): $$(BUILD_DIR)/$$($(1)_BIN)

clean_$(1):
	rm -f  $$($(1)_O_FILES)
	rm -f  $$($(1)_D_FILES)

.PHONY: clean_$(1) $(1)

endef

```

Figure 6: **Generic Makefile Template to Build Any Benchmark.**

This “`Makefile.mk`” fragment is included in the main “`Makefile`” as follows:

```makefile
# All benchmarks/applications
-include $(SRC_DIR)/Makefile.mk
```

If we inspect the “`Makefile.mk`” for the “`vvadd`” benchmark, we can see that it instantiate a generic template by specifying the benchmark name and intended executable name. Extra rules for more targets can be added alongside the template instantiation; however, extra care has to be made to guarantee unique targets for all rules (i.e., the template instantiation defines multiple rules for some targets, which might not be very clear which rules are defined for what targets). Although the generic template can be used and it would work for most common builds for benchmarks, a more customized Makefile can be made instead of instantiating the generic template. The current “`Makefile.mk`” for the “`vvadd`” benchmark is as follows:

```makefile
# Makefile directory
APP_NAME:=$(notdir $(shell dirname $(realpath $(lastword $(MAKEFILE_LIST)))))
$(APP_NAME)_name := $(APP_NAME)
$(APP_NAME)_dir  := $(shell dirname $(realpath $(lastword $(MAKEFILE_LIST))))

# Instantiate the template
$(eval $(call template_mk,$(APP_NAME),$($(APP_NAME)_dir)))
```

```c
bool help = false;
for (int i = 1; i < argc; i++) {
  /* Implementations */
  if (strcmp(argv[i], "-i") == 0 || strcmp(argv[i], "--impl") == 0) {
    assert (++i < argc);
    if (strcmp(argv[i], "naive") == 0) {
      impl = impl_scalar_naive_ptr; impl_str = "scalar_naive";
    } else if (strcmp(argv[i], "opt"  ) == 0) {
      impl = impl_scalar_opt_ptr  ; impl_str = "scalar_opt"  ;
    } else if (strcmp(argv[i], "vec"  ) == 0) {
      impl = impl_vector_ptr      ; impl_str = "vectorized"  ;
    } else if (strcmp(argv[i], "para" ) == 0) {
      impl = impl_parallel_ptr    ; impl_str = "parallelized";
    } else {
      impl = NULL                 ; impl_str = "unknown"     ;
    }

    continue;
  }

  /* Input/output data size */
  if (strcmp(argv[i], "-s") == 0 || strcmp(argv[i], "--size") == 0) {
    assert (++i < argc);
    data_size = atoi(argv[i]) * sizeof(int);

    continue;
  }
  /* Run parameterization */
  if (strcmp(argv[i], "--nruns") == 0) {
    assert (++i < argc);
    nruns = atoi(argv[i]);

    continue;
  }

  if (strcmp(argv[i], "--nstdevs") == 0) {
    assert (++i < argc);
    nstdevs = atoi(argv[i]);

    continue;
  }

  /* Parallelization */
  if (strcmp(argv[i], "-n") == 0 || strcmp(argv[i], "--nthreads") == 0) {
    assert (++i < argc);
    nthreads = atoi(argv[i]);

    continue;
  }

  if (strcmp(argv[i], "-c") == 0 || strcmp(argv[i], "--cpu") == 0) {
    assert (++i < argc);
    cpu = atoi(argv[i]);

    continue;
  }

  /* Help */
  if (strcmp(argv[i], "-h") == 0 || strcmp(argv[i], "--help") == 0) {
    help = true;

    continue;
  }
}

```

Figure 7: **Code for Argument Parsing**—The code shows a for-loop that performs argument parsing. Each if-statement parses one supported argument. If the

The template will build the "`main.c`" of the benchmark; automatically through GNU GCC's "`-MMD`", the template will pick all correct dependencies and build all required object files for the benchmark executable. Please, spend some effort to go over the template and further understand how it works, shown in Figure 6. The "`main.c`" includes code to parse the arguments; arguments are set by the user and passed to the executable (i.e., the program) through the commandline. Figure 7 shows the argument parsing code in "`main.c`".

## 2.4 YABMS Benchmarks: `template`

To facilitate adding new benchmarks, YABMS includes a "`template`" benchmark. To add a new benchmark "`my_new_benchmark`" to YABMS, start by making a copy of the directory "`template`" and call it "`my_new_benchmark`". Then, you can begin to modify the template to implement whatever functionality required. Pay attention to the following: `arguments`, `impls`, `data allocation`, and `checking and testing`. The functions that would ultimately perform the algorithm of the benchmark is implemented as a new `impl` and not in `main.c`.

**Modifying the Argument Structure Type**–start by modifying the default argument structure type (which can be found inside "`include/types.h`") to reflect the intended arguments and their types to all function calls for your benchmark. It is fine if you did not get all the arguments correct from the beginning, often, this is an iterative process; normally, as you develop your benchmark, you may require to modify the structure to add more arguments and/or modify their types. The default argument structure provided by the template is as follows:

```c
#ifndef __INCLUDE_TYPES_H_
#define __INCLUDE_TYPES_H_

typedef struct {
  byte*   input;
  byte*   output;

  size_t size;

  int     cpu;
  int     nthreads;
} args_t;

#endif //__INCLUDE_TYPES_H_
```

By default, the argument structure "`args_t`" includes a byte memory pointer for input, a byte memory pointer for output, and a size variable. The other two arguments (i.e., `cpu` and `nthreads`) can be used to enable passing scheduling parameters to the implementation. This structure should be used for *all* implementations.

**Adding the Correct Implementation**–to modify an implementation to the benchmark, you must modify the correct file in the freshly-made copy "`my_new_benchmark`" of "`template`". Inside the "`impl`" directory, you can see a list of "`.c`" and "`.h`" files. The "`.c`" files are C code files used for implementation and "`.h`" are C header files used for declaration. The header files are included in "`main.c`" to enable correct lazy linking by the GNU linker. The way the implementations are coded, it is unlikely you would need to modify the header files for any implementation; however, if you change the signature of your implementation's function, make sure to change the respective header file accordingly. To add a new implementation, you can create a C code file and a C header file for the implementation. Both C code file and C header file should have the same name. **Do NOT forget to add an include statment for your new header file in "`main.c`"**. Also, do not forget to modify the argument parsing code to recognize the name of the implementation and assign the correct function pointer.

The C header file for the default naïve implementation is as follows:

```c
#ifndef __IMPL_NAIVE_H_
#define __IMPL_NAIVE_H_

/* Function declaration */
void* impl_scalar_naive(void* args);

#endif //__IMPL_NAIVE_H_
```

The corresponding C code file for the naïve implementation:

```c
1  /* Standard C includes */
2  #include <stdlib.h>
3
4  /* Include common headers */
5  #include "common/macros.h"
6  #include "common/types.h"
7
8  /* Include application-specific headers */
9  #include "include/types.h"
10
11 /* Naive Implementation */
12 #pragma GCC push_options
13 #pragma GCC optimize ("O1")
14 void* impl_scalar_naive(void* args)
15 {
16   return NULL;
17 }
18 #pragma GCC pop_options
```

Notice how the type definition for the argument structure is included in line 9. You can see that the signature of the function was used in the header file. Moreover, in lines 5 and 6, the implementation is including the libraries, functions, and macros provided by the benchmark suite.

## 3 Adding a New Benchmark: `mmult`

You are tasked with adding a new benchmark in YABMS, which is `mmult`. This section explains the matrix-matrix multiplication operation. Then, the section will help position you in the right direction to add a `mmult` as a new benchmark in YABMS.

The rest of the section is structured as follows: Subsection 3.1 explains mathematically the matrix-matrix operation; Subsection 3.2 shows a pseudo-code of a naïve implementation of the `mmult` benchmark.

### 3.1 Matrix-Matrix Multiplication Operation

To understand the matrix-matrix multiplication operation, let us explain the matrix-vector multiplication operation first. Let us assume we have a matrix $A$ with dimensions $M \times N$, and a vector $v$ with dimensions $N \times 1$. We can represent matrix $A$ and vector $v$ mathematically as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} \qquad v = \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix}$$

The mathematical definition for matrix-vector multiplication can be defined as:

$$A.v = \begin{bmatrix} a_{11} & a_{12} & . & . & . & a_{1n} \\ a_{21} & a_{22} & . & . & . & a_{2n} \\ . & . & . & & & . \\ . & . & & . & & . \\ . & . & & & . & . \\ a_{m1} & a_{m2} & . & . & . & a_{mn} \end{bmatrix} . \begin{bmatrix} v_1 \\ v_2 \\ . \\ . \\ . \\ v_n \end{bmatrix} = \begin{bmatrix} a_{11}.v_1 + a_{12}.v_2 + ... + a_{1n}.v_n \\ a_{21}.v_1 + a_{22}.v_2 + ... + a_{2n}.v_n \\ . \\ . \\ . \\ a_{m1}.v_1 + a_{m2}.v_2 + ... + a_{mn}.v_n \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \\ . \\ . \\ . \\ r_n \end{bmatrix} = r$$

As we can see, the result of multiplying matrix $A$ by vector $v$ would be a vector $r$ with dimensions $M \times 1$. We can also observe that the second dimension of $A$ must have the same size as the first dimension of the vector $v$. Each element of vector $r$ can be represented with the following equation:

$$r_x = \sum_{i=1}^{N} a_{xi} \times v_i$$

Now, let us consider two matrices $A$ and $B$. Matrix $A$ has dimensions of $M \times N$, while matrix $B$ has dimensions of $N \times P$. We can represent both matrices mathematically as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \qquad B = \begin{bmatrix} b_{11} & b_{12} & \cdot & \cdot & \cdot & b_{1p} \\ b_{21} & b_{22} & \cdot & \cdot & \cdot & b_{2p} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ b_{n1} & b_{n2} & \cdot & \cdot & \cdot & b_{np} \end{bmatrix}$$

Let us assume we want to calculate the matrix-matrix multiplication $A.B$. The definition of the matrix-matrix multiplication begins by decomposing matrix $B$ into multiple vectors as follows:

$$B = \begin{bmatrix} b_{11} & b_{12} & \cdot & \cdot & \cdot & b_{1p} \\ b_{21} & b_{22} & \cdot & \cdot & \cdot & b_{2p} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ b_{n1} & b_{n2} & \cdot & \cdot & \cdot & b_{np} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \cdot \\ \cdot \\ \cdot \\ b_{n1} \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \\ \cdot \\ \cdot \\ \cdot \\ b_{n2} \end{bmatrix} ... \begin{bmatrix} b_{1p} \\ b_{2p} \\ \cdot \\ \cdot \\ \cdot \\ b_{np} \end{bmatrix} \end{bmatrix}$$

The matrix-matrix multiplication, as a result, can be defined as follows:

$$\begin{aligned} A.B &= \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdot & \cdot & \cdot & b_{1p} \\ b_{21} & b_{22} & \cdot & \cdot & \cdot & b_{2p} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ b_{n1} & b_{n2} & \cdot & \cdot & \cdot & b_{np} \end{bmatrix} \\ &= \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \cdot \\ \cdot \\ \cdot \\ b_{n1} \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \\ \cdot \\ \cdot \\ \cdot \\ b_{n2} \end{bmatrix} \dots \begin{bmatrix} b_{np} \\ b_{2p} \\ \cdot \\ \cdot \\ \cdot \\ b_{np} \end{bmatrix} \end{bmatrix} \\ &= A. \begin{bmatrix} \begin{bmatrix} b_{11} \\ b_{21} \\ \cdot \\ \cdot \\ \cdot \\ b_{n1} \end{bmatrix} \begin{bmatrix} b_{12} \\ b_{22} \\ \cdot \\ \cdot \\ \cdot \\ b_{n2} \end{bmatrix} \dots \begin{bmatrix} b_{np} \\ b_{2p} \\ \cdot \\ \cdot \\ \cdot \\ b_{np} \end{bmatrix} \end{bmatrix} \\ &= \begin{bmatrix} A. \begin{bmatrix} b_{11} \\ b_{21} \\ \cdot \\ \cdot \\ \cdot \\ b_{n1} \end{bmatrix} & A. \begin{bmatrix} b_{12} \\ b_{22} \\ \cdot \\ \cdot \\ \cdot \\ b_{n2} \end{bmatrix} & \dots & A. \begin{bmatrix} b_{np} \\ b_{2p} \\ \cdot \\ \cdot \\ \cdot \\ b_{np} \end{bmatrix} \end{bmatrix} \end{aligned} \tag{1}$$

We already defined matrix-vector multiplication. As a result, we can define the matrix-matrix multiplication operation as follows:

$$
\begin{aligned}
A.B &= \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & \cdot & a_{2n} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ a_{m1} & a_{m2} & \cdot & \cdot & \cdot & a_{mn} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdot & \cdot & \cdot & b_{1p} \\ b_{21} & b_{22} & \cdot & \cdot & \cdot & b_{2p} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ b_{n1} & b_{n2} & \cdot & \cdot & \cdot & b_{np} \end{bmatrix} \\
&= \begin{bmatrix} a_{11}.b_{11} + \cdots + a_{1n}.b_{n1} & a_{11}.b_{12} + \cdots + a_{1n}.b_{n2} & \cdot & \cdot & \cdot & a_{11}.b_{1p} + \cdots + a_{1n}.b_{np} \\ a_{21}.b_{11} + \cdots + a_{2n}.b_{n1} & a_{21}.b_{12} + \cdots + a_{2n}.b_{n2} & \cdot & \cdot & \cdot & a_{21}.b_{1p} + \cdots + a_{2n}.b_{np} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ a_{m1}.b_{11} + \cdots + a_{mn}.b_{n1} & a_{m1}.b_{12} + \cdots + a_{mn}.b_{n2} & \cdot & \cdot & \cdot & a_{m1}.b_{1p} + \cdots + a_{mn}.b_{np} \end{bmatrix} \\
&= \begin{bmatrix} r_{11} & r_{12} & \cdot & \cdot & \cdot & r_{1p} \\ r_{21} & r_{22} & \cdot & \cdot & \cdot & r_{2p} \\ \cdot & \cdot & \cdot & & & \cdot \\ \cdot & \cdot & & \cdot & & \cdot \\ \cdot & \cdot & & & \cdot & \cdot \\ r_{m1} & r_{m2} & \cdot & \cdot & \cdot & r_{mp} \end{bmatrix} \\
&= R
\end{aligned}
\tag{2}
$$

We can see that the result of multiplying matrices $A$ and $B$ is a matrix $R$ with dimensions $M \times P$. The elements of the resulting matrix $R$ is defined mathematically as follows:

$$r_{xy} = \sum_{i=1}^{N} a_{xi} \times b_{iy}$$

For the matrix-matrix multiplication to be possible, the size of the second dimension of matrix $A$ must be the same as the size of the first dimension of matrix $B$.

## 3.2 Naïve Implementation of `mmult`

The pseudo-code of a naïve implementation of `mmult` calculates each element $r_{xy}$ of the resulting matrix $R$ to completion. The high-level pseudo-code of a naïve implementation of `mmult` is shown in Algorithm 1; the naïve implementation assumes two input matrices $A$ (with dimensions $M \times N$) and $B$ (with dimensions $N \times P$). The element-type of the matrices should be IEEE754 floating-point 32-bit single, which stresses the hardware more than integers.

## 3.3 Datasets and Golden Reference

As part of the benchmark implementation, five randomized datasets should be included: test, small, medium, large, and native, detailed in Table 1. The datasets can be used to test and debug your implementation. Also, the datasets can be used to evaluate performance. Each dataset contains matrix $A$, matrix $B$, and their resultant matrix of the multiplication $R$ (where $R = A \times B$). After executing an implementation, the benchmark should provide the ability to check the implementation's output against the golden reference from the dataset (i.e., matrix $R$). The verification logic requires intricate consideration; comparing integers, for example, is vastly different than comparing logical values or floating-point values. For floating-point values, in specific, a comparison is often performed by the value to-be-compared from the reference value; then, the difference is checked against a predetermined threshold. There are other methods that can be used to estimate overall error instead of individual error (e.g., mean squared error); such methods can be used only if it is appropriate for the benchmark. The decision depends on whether the benchmark execution is defined by the overall error or the individual error instead.

Table 1: **Datasets Information**–table showing dimensions for input datasets and the golden reference.

| Dataset | Matrix A | | Matrix B | | Matrix R | |
|---|---|---|---|---|---|---|
| | M | N | M | N | M | N |
| testing | 16 | 12 | 12 | 8 | 16 | 8 |
| small | 121 | 180 | 180 | 115 | 121 | 115 |
| medium | 550 | 620 | 620 | 480 | 550 | 480 |
| large | 962 | 1,012 | 1,012 | 1,221 | 962 | 1,221 |
| native | 2,500 | 3,000 | 3,000 | 2,100 | 2,500 | 2,100 |

There are multiple ways to handle generating the dataset and the golden reference; the naïve way is to generate the inputs from the dataset using a randomized routine in the benchmark. The golden reference is computed by using the reference implementation with the randomly generated inputs. The naïve method of handling the dataset and the golden reference is easy to implement, but it is susceptible to bugs in the reference implementation. Moreover, the overhead of generating the dataset and the golden reference scales with the requested size. A more methodical way is to generate a randomized input dataset, then a more verified implementation can be used to generate the golden reference. These input dataset and golden reference is, then, stored in files. The files are then loaded into the benchmark. The verification

**Algorithm 1** Naïve `mmult` Implementation

```
1: procedure MMULT($A, B$)
2:     $R \leftarrow empty\_matrix(M, P)$
3:     for $i = 1 \rightarrow M$ do
4:         for $j = 1 \rightarrow P$ do
5:             $R[i][j] \leftarrow 0$
6:             for $k = 1 \rightarrow N$ do
7:                 $R[i][j]$ += $A[i][k]$ * $B[k][j]$
8:             end for
9:         end for
10:    end for
11:    return $R$
12: end procedure
```

## 4 Deliverables and Report

The work required for this assignment is divided into two parts: deliverables, and a report.

### 4.1 Deliverables

There are two deliverables for this assignment. These deliverables and their explanations are as follows:

1. **Benchmark Implementation in C:** The naïve implementation as a part of the `mmult` benchmark, which is described in Section 3. One can leverage the already-implemented `vvadd` benchmark in YABMS to draw some insipration about implementing the `mmult` benchmark. The pseudo-code, shown in Algorithm 1, can be used as a reference.

2. **Test Cases and Results:** Some kind of testing and verification performed on the implementation to verify its execution. These test cases are expected to span multiple values of matrix dimensions and they are expected to stress the implementation and verify its execution when compared to a golden reference execution (e.g., MATLAB).

### 4.2 Report

Each student who worked on the assignment is expected to submit a report. This report details the students work on the deliverables and discusses their observations and findings. The bulk of grading will be based on the report portion of the submission. Each group should make sure to allocate enough time to polish their report as it would reflect their effort and work. The report is expected to have the following sections:

1. **Introduction:** A section that introduces the problem and the goal of the report. The section should summarize the student's contributions and key observations. The section should include a paragraph summarizing the student's understanding of the matrix-matrix multiplication operation.

2. **Collaboration:** Since students are allowed to discuss the baseline implementation, and the alternative implementation, the report should include a section that would details the collaborations between the student and other students. The section should mention other students by name and what aspect was discussed with them. **Make sure that collaboration is NOT allowed while implementing any code OR writing the report. Each student is ONLY allowed to collaborate with other students by discussing improvements and implementation aspects, but not collaborate during the actual implementation or writing the report.**

3. **Naïve Implementation:** A section that discusses the baseline scalar and single-threaded implementation of the baseline algorithm. The student is expected to discuss implementation aspects and details their experience. The section should discuss whether there were bugs and how the student would have resolved them. The section should also discuss insights and observations.

4. **Evaluation:** A section that discusses the evaluation methodology and results. The section should details the evaluation methodology and justify it. Once the methodology is outlined and justified, the section should details the evaluation results and present these results in either tables or figures (whatever is appropriate). It is unacceptable for this part of evaluation to be devoid of any tables or figures. One should place a lot of thought into some interesting experiments that can be conducted to highlight some interesting details. The section should conclude with results analysis and takeaways.

5. **Conclusion:** A section that summarizes the students observations, insights, and findings. This section should also include any lessons learned by doing the assignment; you can also discuss any difficulties.

6. **References:** A section with citation to sources and material that helped in the assignment.

## References

[1] GNU Not Unix (GNU). *GNU `make` Utility*. `https://www.gnu.org/software/make/manual/make.html`. Accessed: 2024-10-02.

[2] Institute of Electrical and Electronics Engineers. *IEEE Standard for Information Technology–Portable Operating System Interfaces (POSIX®)–Part 2: Shell and Utilities*. `https://ieeexplore.ieee.org/servlet/opac?punumber=6880749`. Accessed: 2024-10-02.

[3] Khalid Al-Hawaj. *YABMS: Yet Another Benchmark Suites*. `https://github.com/hawajkm/YABMS`. Accessed: 2025-02-21.