Friedrich-Alexander-Universität
Technische Fakultät

COMPUTER ARCHITECTURE
INFIII

Computer Architecture Practical Exercise

1 Vector Summation

Kenan Gündogan[1] **Philipp Gündisch**[1]

[1]Friedrich-Alexander Universität Erlangen-Nürnberg, Chair of Computer Science 3 (Computer Architecture)

November 4, 2024

Benchmarking

Principles

What are good properties of a benchmark?

Benchmarking

Principles

- **Relevance**
  Benchmarks should measure relatively vital features.
- **Representativeness**
  Benchmark performance metrics should be broadly accepted by industry and academia.
- **Equity**
  All systems should be fairly compared.
- **Repeatability**
  Benchmark results can be verified.
- **Cost-effectiveness**
  Benchmark tests are economical.
- **Scalability**
  Benchmark tests should work across systems possessing a range of resources from low to high.
- **Transparency**
  Benchmark metrics should be easy to understand.

Principles taken from Benchmarking Contemporary Deep Learning Hardware and Frameworks.

Benchmarking

Principles Takeaways

- **Relevance**
  Benchmarks should measure relatively vital features $\rightarrow$ defined by exercises.
- **Representativeness**
  Benchmark performance metrics should be broadly accepted by industry and academia.
- **Equity**
  All systems should be fairly compared.
- **Repeatability**
  Benchmark results can be verified $\rightarrow$ repeated execution produces same result.
- **Cost-effectiveness**
  Benchmark tests are economical $\rightarrow$ allocate only needed amount of resources.
- **Scalability**
  Benchmark tests should work across systems possessing a range of resources from low to high.
- **Transparency**
  Benchmark metrics should be easy to understand $\rightarrow$ keep it as simple as possible.

Not all principles are relevant for this course since we want to understand the effect of coding techniques on the dedicated cluster rather than defining a new standard for an industry wide benchmark.

Benchmarking

Relevance

In this exercise we will implement the `vec_sum()` function which sums up an array of floating point values.

- The processed *additions per second* are a good performance indicator
- This measure gives insights about the adder unit of the benchmarked processor

Benchmarking

Repeatability

- A benchmark should always produce the *"same"* results.
- We define results as the same if and only if the same conclusions can be derived from those results.
- How can we achieve this repeatability?
  - Minimization of runtime fluctuations
  - Robust benchmark design
- Causes for inconsistent results?
  - Varying clock frequencies
  - Interfering threads and processes
  - System calls
  - Unpinned threads
  - Thread synchronization
  - Inconsistent tooling (Compiler and Flags, ...)
  - e.t.c

Benchmarking

Cost-Effectiveness

- Calculate the theoretical worst case runtime of the benchmark in advance
- The runtime of a benchmark consists of
  - Slurm launch overhead
  - Program execution time
  - (optionally) Program compile time
  - Whatever you do in the shell script ...
- Adjust the `--time` parameter in the sbatch script

Benchmarking

Transparency

The benchmark metrics will be discussed in the exercises such that a common understanding is guaranteed.

Shell Script

Update

To reduce runtime fluctuations we instruct Slurm with additional `sbatch` flags.

```
#/bin/bash -l

#SBATCH --partition=singlenode
#SBATCH --time=00:05:00
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --exclusive
#SBATCH --cpu-freq=2400000-2400000:performance
#SBATCH --export=NONE

...
srun ./program
```

- `--exclusive` suppressing other users to start programs on allocated node
- `--cpu-freq` specifying the allowed CPU frequency range

Makefile

To reduce inconsistencies in the toolchain we recommend to use `make` for the build process. Therefore, we provide a Makefile for you to start with:

```
.PHONY: all clean vecSum

ROOT_PATH  := .
SRC_PATH   := $(ROOT_PATH)/src
BUILD_PATH := $(ROOT_PATH)/build
BIN_PATH   := $(ROOT_PATH)/bin

INC_PATH   := $(SRC_PATH)/include
INC_DIRS   := $(sort $(shell find $(INC_PATH) -type d))
INC_FLAGS  := $(addprefix -iquote ,$(INC_DIRS))

CC         := icx
CFLAGS     := -Wall -pedantic -Werror -std=c99 -O3 -xHost
LDFLAGS    :=

all: vecSum # program is named vecSum

clean:
	rm -rf $(BUILD_PATH) $(BIN_PATH)

vecSum: $(BIN_PATH)/vecSum

$(BIN_PATH)/vecSum: $(BUILD_PATH)/main.o $(BUILD_PATH)/vec_sum.o $(BUILD_PATH)/get_time.o
	mkdir -p $(dir $@)
	$(CC) $(CFLAGS) $^ -o $@ $(LDFLAGS)

$(BUILD_PATH)/%.o: $(SRC_PATH)/%.c
	mkdir -p $(dir $@)
	$(CC) $(INC_FLAGS) $(CFLAGS) -MMD -MP -c $< -o $@
```

Note: Feel free to adjust the makefile according to your needs

Exercise 1.1: Implement Vector Summation

- Implement a function `float vec_sum(const float * restrict array, int32_t length)` in `src/vec_sum.c` which adds up all elements of an array and returns the sum
- Complete the main function in `src/main.c`
  - Command line parameters:
    - Size of array in KiB
    - Minimal runtime for the benchmark in milliseconds
- Allocate memory for an array with `malloc`
- Repeatedly call `vec_sum()` until the minimal runtime *(see main.c)* is exceeded
  - Use `get_time()` for time measurement, which is provided in the file `src/get_time.c`
- Compile the program with the Intel Compiler (`icx`) using the following flags: `-O3 -xHost` *(see Makefile)*
- What is the theoretical worst case execution time of your program when 1ms is specified?

Exercise 1.2: Reasonable Runtime

- Design a benchmark to identify a suitable minimal runtime
- Implement the newly provided `cluster.sh`
- Run the benchmark with `sbatch`
- What is the optimal minimal runtime according to your `result.csv` and the discussed principles?

Exercise 1.3: Performance

- Duplicate the `cluster.sh` from 1.2 and update it for the performance benchmark
- Benchmark your implementation for 1KiB - 32MiB (with 1 second minimal runtime)
- How long does your program (single execution) actually run in the worst case?
- Determine the maximum benchmark runtime and adapt the `sbatch --time` argument accordingly
- Run the benchmark with `sbatch`
- For what array size is the best performance achieved?

Tasks

- E 1.1: Implement Vector Summation
  - Implement on head node
  - Work on compute-node only for benchmarking and testing
  - Implement `src/main.c` and `src/vec_sum.c`
  - Implement `scripts/cluster.sh`
  - Makefile is provided
  - Run `sbatch scripts/cluster.sh` on cluster node
  - Save the `result.csv`
- E 1.2: Reasonable Runtime
  - Adapt `scripts/cluster.sh` for time measurements
  - Run benchmark on cluster node and analyze results
  - Identify optimal minimal runtime
- E 1.3: Performance Benchmark
  - Duplicate `scripts/cluster.sh` and adapt for performance measurement
  - Run benchmark on cluster node and analyze results
  - Identify configuration for maximum performance

Appendix: Checklist

Performance Optimization

During the timeline of this class new bullet points will be added.

- Compiling
  - Choice of the compiler (`icx`)
  - Compiler flag to optimize aggressively (e.g. `-O3`)
  - Compiler flag to adapt for specific hardware (e.g. `-xHost`)
- Programming Techniques (if applicable)
  - Use `#define` and `const` instead of variables
  - Data type aware programming
  - Consecutive address iteration
- Measurement
  - Reasonable benchmark time
  - Reasonable benchmark workload
  - Reduce interference factors to a minimum