Chair of Scientific Computing in Computer Science (SCCS)
TUM School of Computation, Information and Technology (CIT)
Technical University Munich

A Practical Introduction to Microbenchmarking

Michael Lettrich

tCSC on Scientific Software for Heterogeneous Architectures

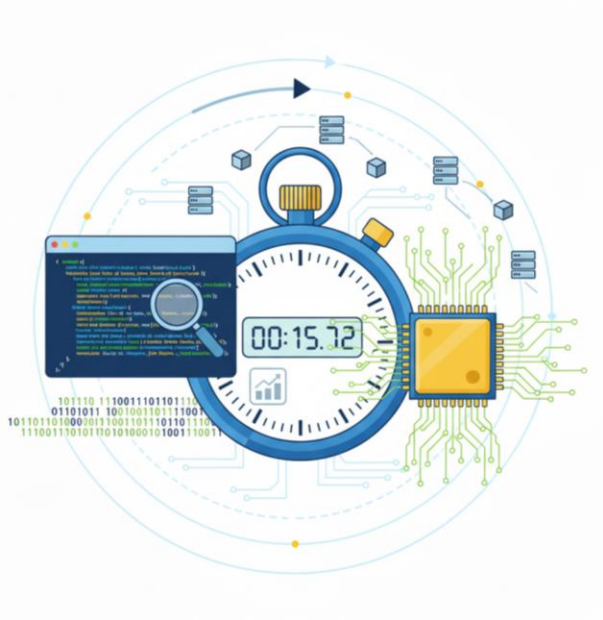
07.10.2025

ALICE

What is a Microbenchmark?

- Experimental tool to track or measure performance of a single, well-defined task.
- Targeted at small to very small building blocks of real programs.

Small, individual application to measure the performance of code in an idealized but controlled execution environment.



When, Why and How to use Microbenchmarks

When

- Part of the performance engineering feedback loop.
- After identification of application hotspots using timing and profilers.

Why

- For deepening understanding of performance behavior.
- To study interactions with underlying hardware.
- When we need a separated, low noise, low overhead environment.

How

- Analyze performance measurements in benchmark.
- Record hardware performance counters with profiler.
- Inspect assembly code.

Google Benchmark for Microbenchmarking

- Open source (micro)benchmark support framework in modern C++
- Available on Github https://github.com/google/benchmark or with your package manager.
- Similar handling to unit testing frameworks like gtest, Boost.Test.
- Quite powerful, we are only going to scratch the surface

DISCLAIMER:
To get meaningful output in microbenchmarking it is important to keep the environment simple without idealizing it too much. These examples focus more on the tool than on methodology.

Schönauer Triad Example

```
1  #include <benchmark/benchmark.h>
2  #include <vector>
3  static void BM_Triad(benchmark::State &state){
4
5
6
7
8
9
10
11
12     for (auto _ : state){ // Benchmark loop
13
14
15
16
17
18     }}
19 BENCHMARK(BM_Triad); // Register the function as a benchmark
20 BENCHMARK_MAIN();
```

Schönauer Triad Example

```cpp
#include <benchmark/benchmark.h>
#include <vector>
static void BM_Triad(benchmark::State &state){
    constexpr size_t N = 1024; // Array size
    std::vector<double> A(N), B(N), C(N);  // Input arrays
    std::vector<double> D(N, 0.0); // Output array
    for (size_t i = 0; i < N; ++i){ // Initialize input data
        A[i] = i * 1.0;
        B[i] = i * 2.0;
        C[i] = i * 3.0;
    }
    for (auto _ : state){ // Benchmark loop



    }}
BENCHMARK(BM_Triad); // Register the function as a benchmark
BENCHMARK_MAIN();
```

Schönauer Triad Example

```cpp
#include <benchmark/benchmark.h>
#include <vector>
static void BM_Triad(benchmark::State &state){
    constexpr size_t N = 1024; // Array size
    std::vector<double> A(N), B(N), C(N);  // Input arrays
    std::vector<double> D(N, 0.0); // Output array
    for (size_t i = 0; i < N; ++i){ // Initialize input data
        A[i] = i * 1.0;
        B[i] = i * 2.0;
        C[i] = i * 3.0;
    }
    for (auto _ : state){ // Benchmark loop
        for (size_t i = 0; i < N; ++i){
            D[i] = A[i] + B[i] * C[i];
        }
        benchmark::DoNotOptimize(D.data()); // Prevent compiler to optimize away D (unused results)
        benchmark::ClobberMemory();         // Memory barrier to prevent compiler optimizations
    }}
BENCHMARK(BM_Triad); // Register the function as a benchmark
BENCHMARK_MAIN();
```

Schönauer Vector Triad – Benchmark Output

```
michael@PCMichi:~/Projects/benchmark-lightning/build$ ./triad0.exe --benchmark_min_warmup_time=1
2025-10-06T22:03:29+02:00
Running ./triad0.exe
Run on (8 X 4900 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 12288 KiB (x1)
Load Average: 0.12, 0.08, 0.08
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may be noisy and will incur extra overhead.
-----------------------------------------------------
Benchmark           Time             CPU   Iterations
-----------------------------------------------------
BM_Triad          320 ns          319 ns      2183706 _
```

Benchmark runs kernel multiple times for meaningful measurements.

Time = "Average Wall Time"

CPU = "Average CPU Time"

Iterations = Number of repetitions

Schönauer Vector Triad – Benchmark Output as JSON

$./triad0.exe --benchmark_format=json --benchmark_out=triad.json

```
"benchmarks": [
  {
    "name": "BM_Triad",
    "family_index": 0,
    "per_family_instance_index": 0,
    "run_name": "BM_Triad",
    "run_type": "iteration",
    "repetitions": 1,
    "repetition_index": 0,
    "threads": 1,
    "iterations": 2875470,
    "real_time": 2.4282841414004912e+02,
    "cpu_time": 2.4278156962166182e+02,
    "time_unit": "ns"
  }
]
```

Schönauer Triad – Custom Metrics

```cpp
for (auto _ : state){
    for (size_t i = 0; i < N; ++i)
        D[i] = A[i] + B[i] * C[i];
    benchmark::DoNotOptimize(D.data());
    benchmark::ClobberMemory();
}
// internal counter for processed items
state.SetItemsProcessed(state.iterations() * N);
// internal counter for number of processed bytes
state.SetBytesProcessed(state.iterations() * N * sizeof(double) * 4);
// Custom counter for array size
state.counters["ArraySize"] = benchmark::Counter(N * sizeof(double),
                                                 benchmark::Counter::kDefaults,
                                                 benchmark::Counter::OneK::kIs1024);
// Custom counter for total memory usage
state.counters["TotalMemory"] = benchmark::Counter(state.counters["ArraySize"] * 4,
                                                   benchmark::Counter::kDefaults,
                                                   benchmark::Counter::OneK::kIs1024);
// Custom FLOPS counter
state.counters["Flops"] = benchmark::Counter(
    N * 2,
    benchmark::Counter::kIsIterationInvariantRate);
```

ALICE

Schönauer Triad – Custom Metrics Output

```
Running ./triad1_metrics.exe
Run on (8 X 4900 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 12288 KiB (x1)
Load Average: 0.02, 0.05, 0.07
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may be noisy and will incur extra overhead.
-----------------------------------------------------------------------------------------------------------------------
Benchmark           Time             CPU   Iterations  ArraySize      Flops TotalMemory bytes_per_second items_per_second
-----------------------------------------------------------------------------------------------------------------------
BM_Triadd         320 ns          319 ns      2160210 _      8Ki 6.41275G/s        32Ki      95.5574Gi/s       3.20637G/s
```

Schönauer Triad – Passing Arguments

```cpp
static void BM_Triad(benchmark::State &state)
{
    const size_t N = state.range(0); // Array size passed externally via state
    // ... other setup code here ...
    for (auto _ : state)
    {
        for (size_t i = 0; i < N; ++i)
        {
            D[i] = A[i] + B[i] * C[i];
        }
        benchmark::DoNotOptimize(D.data()); // Prevent compiler to optimize away D (unused results)
        benchmark::ClobberMemory();         // Memory barrier to prevent compiler optimizations
    }
    // .. custom metrics here ...
}
BENCHMARK(BM_Triad)->Args({1<<10})->Args({1<<11})->Args({1<<12}); // 1K, 2K, 4K elements
BENCHMARK(BM_Triad)->ArgsProduct({benchmark::CreateRange(1<<3, 1<<27, 2)}); // 8 to 128M elements in powers of 2
BENCHMARK_MAIN();
```

Run for different input sizes: Either specify manually or generate entire ranges.

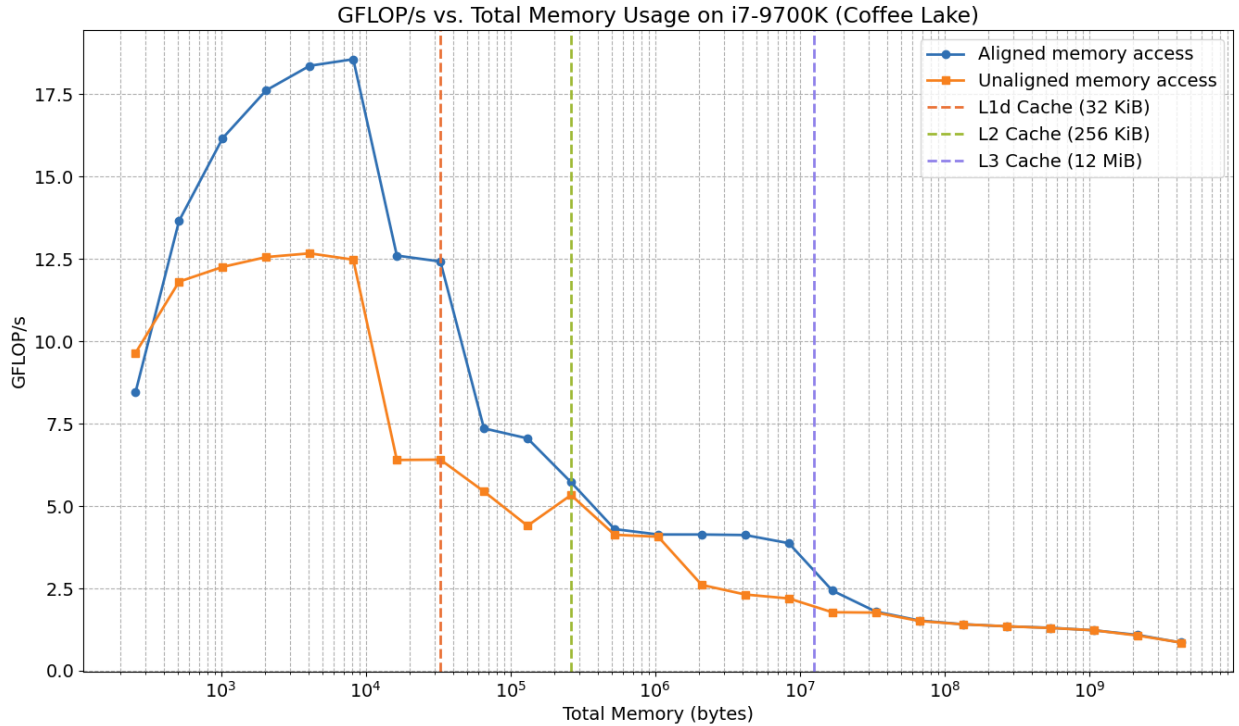
ALICE

Schönauer Triad – Passing Arguments Output

Benchmark	Time	CPU	Iterations	ArraySize	Flops	TotalMemory	bytes_per_second	items_per_second
BM_Triad_aligned/8	1.89 ns	1.89 ns	367487644	64	8.45156G/s	256	125.938Gi/s	4.22578G/s
BM_Triad_aligned/16	2.35 ns	2.34 ns	294203584	128	13.6562G/s	512	203.493Gi/s	6.82809G/s
BM_Triad_aligned/32	3.96 ns	3.96 ns	176639371	256	16.1641G/s	1Ki	240.863Gi/s	8.08203G/s
BM_Triad_aligned/64	7.27 ns	7.27 ns	95617482	512	17.6182G/s	2Ki	262.532Gi/s	8.80911G/s
BM_Triad_aligned/128	13.9 ns	13.9 ns	50379731	1Ki	18.363G/s	4Ki	273.629Gi/s	9.18148G/s
BM_Triad_aligned/256	27.6 ns	27.6 ns	25681504	2Ki	18.5626G/s	8Ki	276.604Gi/s	9.28128G/s
BM_Triad_aligned/512	81.3 ns	81.3 ns	8660084	4Ki	12.6023G/s	16Ki	187.789Gi/s	6.30115G/s
BM_Triad_aligned/1024	165 ns	165 ns	4266315	8Ki	12.4222G/s	32Ki	185.105Gi/s	6.2111G/s
BM_Triad_aligned/2048	557 ns	557 ns	1268371	16Ki	7.35704G/s	64Ki	109.628Gi/s	3.67852G/s
BM_Triad_aligned/4096	1161 ns	1161 ns	598437	32Ki	7.05431G/s	128Ki	105.117Gi/s	3.52716G/s
BM_Triad_aligned/8192	2864 ns	2863 ns	248242	64Ki	5.72193G/s	256Ki	85.2634Gi/s	2.86097G/s
BM_Triad_aligned/16384	7624 ns	7622 ns	91913	128Ki	4.29917G/s	512Ki	64.0626Gi/s	2.14958G/s

ALICE

Schönauer Triad – Bandwidth vs Total Memory Usage



More Features

- Templated benchmarks (run for different types)
- Set up and tear down functions for complex initialization and clean up (DB connections, sockets, files)
- Custom timing functions (e.g. for GPU support)
- Multithreading support (including custom handlers for threading frameworks like OpenMP or TBB)
- Profiler API support (e.g. Intel VTune API, Score-P)
- And more https://google.github.io/benchmark/user_guide.html

More Resources

- Courses on Node Level Performance Engineering https://blogs.fau.de/hager/talks/nlpe
- On timing very short programs https://youtu.be/Czr5dBfs72U?si=jXFMCMCyMj88U8fT
- Almost all talks by Chandler Carruth: https://youtu.be/nXaxk27zwlk?si=zzqRBtkoEu7YK13b