Erlangen Regional Computing Center

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

# A HPM based Methodology for SIMD Code Analysis

High Performance Computing

# Basic procedure for analyzing hot spot

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Instrument hot spot and measure HPM counters for 1 process running on one core with fixed frequency.

1. Ensure code is limited by instruction throughput
2. Measure executed instruction counts for the following categories:
   - FP arithmetic scalar
   - FP arithmetic SIMD (different flavors)
   - Load and store
   - Branch
   - Total
3. Measure CPI/IPC to quantify execution efficiency
4. Compute relations between counts

**Do this for scalar and all SIMD variants you are interested in.**
(to disable autovectorization with Intel compilers use **`-no-vec`**)

# Analyzing instruction decomposition

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

1. Percentages of **category instruction counts** compared to total number of instructions
2. Factor of **total and arithmetic instruction count** for SIMD variants vs scalar baseline

**Outcome**

- For a code where algorithmic work consists of mostly floating point computations the **instruction decomposition** should be dominated by **arithmetic** floating point **instructions**.

- The **SIMD factor**, means the number of lanes in a SIMD register should be visible for overall as well as arithmetic instruction counts. SIMD saves instructions but also often requires to add new instructions to enable SIMD processing.

# Analyzing execution efficiency

Compare **CPI** and **runtime** for all SIMD variants  vs. scalar baseline.

**Outcome**

- CPI quantifies superscalar execution. Instruction count times CPI gives runtime in cycles.

- The runtime factor should be equal to the **#instructions X CPI**

# Benefits of method

- Algorithmic and SIMD benefits can be quantified on an **instruction level**
- It can be quantified how well the SIMD vectorization works separated for **instruction level** and **execution side**.
- Overheads can be named, most effective together with static assembly code analysis:
  - Overhead added getting data from memory into registers
  - Overhead added by compiler for implementing programming language or programming model
  - Identify potential to save instructions

# Example: MiniMD proxy app

Proxy app from **MANTEVO**.

**`https://mantevo.org`**


- **`MiniMD`** – MPI + OpenMP
  - App. 3000 lines, C++
  - Extracted from LAMMPS (**`http://lammps.sandia.gov`**) Molecular dynamics application code
  - Other implementations: OpenACC, OpenCL, Kokkos
  - Supports Lennard-Jones (default) and eam force calculation
  - Uses neighbour lists

# How to compile and run

- Load modules for compiler and MPI
- Edit **`include_ICC.mk`** for compilation options and flags

- Compile with: **`$ make`**
- Clean up with: **`$ make clean`**

- You have to chdir to the ./data directory to run the code:
  **`$ cd ./data`**

- Run the benchmark by calling:
  **`$ ../miniMD-ICC --half_neigh <0|1> -n 200`**