Contents

1 Benchmarks and results

All the source codes of the internship are available in this repository. We provide a set of executables to test the different versions presented above. For executables that use Kokkos, it is important to note that they must be compiled with a Kokkos installation configured for the desired **backend** (e.g., CUDA, OpenMP, or C++ threads...). As briefly mentioned before, we used Guix to manage our development environments and ensure reproducibility. Since Kokkos configurations need to be set at **compile time**, several variants of the Kokkos package were introduced in Guix (*e.g.*, `kokkos-openmp`, `kokkos-cuda-a100`, ...). For reproducibility, we provide a dedicated **channel file** to leverage the `guix time-machine` command. Additionally, we created isolated development environments using the `guix shell` command combined with the `--pure` option that unsets existing **environment variables** to prevent any "collision" with external packages. Although the `--container` option would provide a fully isolated environment, it is not currently supported on PlaFRIM due to an outdated version of the running **CentOS** kernel, preventing **containerization**. Consequently, we only relied on the `--pure` option.

We present various experiments that can be run across different **backends**. For each binary, a list of available options can be viewed using the `--help` option as follows:

```
./excutable --help
```

1.1 On the CPU (using OpenMP)

First, we set-up the environment via Guix:

```
guix time-machine -C guix-tools/channels.scm -- shell --pure
↪  -m guix-tools/manifest-openmp.scm -- bash --norc
```

We get all the executables at once with these **CMake** commands:

```
rm -rf build-non-cuda && cmake --preset release-non-cuda
cmake --build --preset build-non-cuda
```

The following command runs the naive (fully **sequential**) version introduced earlier to compute interactions between 1,000 randomly generated particles in single precision, with the computation repeated 5 times in a row.

```
build-non-cuda/bench-ref --number-runs 5 --number-of-particles
↪  1000 --use-float --verbose
```

As recommended by the documentation of Kokkos, we can set the `KOKKOS_PRINT_CONFIGURATION` environment variable to enable verbose output and display the current configuration.

```
export KOKKOS_PRINT_CONFIGURATION=1
```

Since, we use OpenMP as a backend, we can use `OMP_PROC_BIND` and `OMP_PLACES` to control how threads are assigned to CPU cores. If these variables are not set, Kokkos will display a warning message prompting you to configure them.

```
export OMP_PROC_BIND=spread
export OMP_PLACES=threads
```

We set `OMP_PROC_BIND=spread` to instruct the OpenMP runtime to distribute threads as evenly as possible across the available cores. Additionally, we configure OpenMP to place threads on separate hardware threads (*i.e.,* logical cores when hyper-threading is enabled).

Finally, as usual, we can set the number of threads using `OMP_NUM_THREADS`, though it is recommended to use `KOKKOS_NUM_THREADS` instead, as it also applies to the C++ threads backend.

```
export KOKKOS_NUM_THREADS=2
```

The different versions can be run with the following commands, where the `--check` option compares the final numerical result with the reference result (baseline version).

```
build-non-cuda/bench-kokkos-flat-parallel-for --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check

build-non-cuda/bench-kokkos-flat-parallel-reduce --number-runs
↪  5 --number-of-particles 1000 --use-float --verbose --check

build-non-cuda/bench-kokkos-nested-parallel --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check

build-non-cuda/bench-kokkos-hierarchical --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check
```

1.2 On the CPU (for vectorization)

For optimal vectorization, Guix allows packages to be marked as **tunable**, which is essential to fully take advantage of SIMD capabilities (as explained in the documentation). When using the `--tune[=arch]` option (typically `--tune=native`), binaries are compiled with the `-march=arch` flag (see x86 Options for examples), enabling them to specifically target the desired CPU architecture. Prior to the internship, the Kokkos packages available in Guix were not marked as **tunable**, so we created a local variant (defined in kokkos-variants.scm) to add this property, as described in the Guix reference manual. The original package is expected to be updated soon, so this custom version will no longer be necessary. As a result, we will be able to use the `--tune[=arch]` option directly. For example, for Intel Xeon Skylake Gold 6240 processors on Bora nodes in PlaFRIM (which support the AVX-512 instruction set), we provide a suitable manifest file that can be used as follows:

```
guix time-machine -C guix-tools/channels.scm -- shell --pure
↪  -m guix-tools/manifest-cpu-skylake.scm -L guix-tools/ --
↪  bash --norc
rm -rf build-non-cuda && cmake --preset release-non-cuda
cmake --build --preset build-non-cuda
```

Once, everything is setup we can run the program using either xsmid:

```
build-non-cuda/bench-xsimd --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check
```

or Kokkos SIMD:

```
build-non-cuda/bench-kokkos-simd --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check
```

1.3 On the GPU (using CUDA)

Similarly, we can also perform experiments using CUDA as a backend. For instance, on a Sirocco node of PlaFRIM equipped with NVIDIA A100 GPUs:

```
guix time-machine -C guix-tools/channels.scm -- shell --pure
↪  -m guix-tools/manifest-a100.scm -- env LD_PRELOAD=/usr/lib
↪  64/libcuda.so:/usr/lib64/libnvidia-ptxjitcompiler.so bash
↪  --norc
rm -rf build-cuda && cmake --preset release-cuda
cmake --build --preset build-cuda
```

We can start by running the baseline CUDA version:

```
build-cuda/bench-cuda-shared --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check
↪  --threads-per-block 256
```

Likewise, for convenience, we set the `KOKKOS_PRINT_CONFIGURATION` variable to enable verbose output and display the current configuration.

```
export KOKKOS_PRINT_CONFIGURATION=1
```

As in the previous cases, the executables can be run as follows:

```
build-cuda/bench-kokkos-flat-parallel-for --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check

build-cuda/bench-kokkos-flat-parallel-reduce --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check

build-cuda/bench-kokkos-nested-parallel --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check

build-cuda/bench-kokkos-hierarchical --number-runs 5
↪  --number-of-particles 1000 --use-float --verbose --check
```

1.4 Run all the experiments

Finally, we provide **batch scripts** to run directly all the experiments on PlaFRIM. This can be achieved using the following commands:

```
sbatch scripts/experiments-batch-bora-openmp.batch
sbatch scripts/experiments-batch-bora-threads.batch
sbatch scripts/experiments-batch-a100.batch
sbatch scripts/experiments-batch-v100.batch
sbatch scripts/experiments-batch-p100.batch
sbatch scripts/experiments-batch-broadwell.batch
sbatch scripts/experiments-batch-skylake.batch
```

1.5 Plots

After running all the experiments from the previous section, we provide a Python script plot-results.py to plot the results.

1.5.1 Comparison: xsimd vs Kokkos SIMD

```
mkdir -p img
python3 scripts/plot-results.py --data-files \
results/bench-xsimd-f64-sirocco15.plafrim.cluster-experiment-s
↪ kylake.txt
↪ \
results/bench-kokkos-simd-f64-sirocco15.plafrim.cluster-experi
↪ ment-skylake.txt
↪ \
results/bench-xsimd-f32-sirocco15.plafrim.cluster-experiment-s
↪ kylake.txt
↪ \
results/bench-kokkos-simd-f32-sirocco15.plafrim.cluster-experi
↪ ment-skylake.txt
↪ \
--colors "yellowgreen" "violet" "lightgreen" "slateblue" \
--labels "xsimd (fp 64)" "kokkos simd (fp 64)" "xsimd (fp 32)"
↪ "kokkos simd (fp 32)" \
--linewidths 2 3 2 3 \
--marker d d s s \
--linestyles "-" ":" "-" ":" \
```

```
--title $'Comparison Kokkos::simd and xsimd\nIntel Xeon
↪  Skylake Gold 6240 @ 2.6 GHz [AVX512]' \
--xlabel "size (N)" \
--ylabel "time (s)" \
--xscale log \
--yscale log \
--plot-file "img/plot-simd-skylake.png"
```

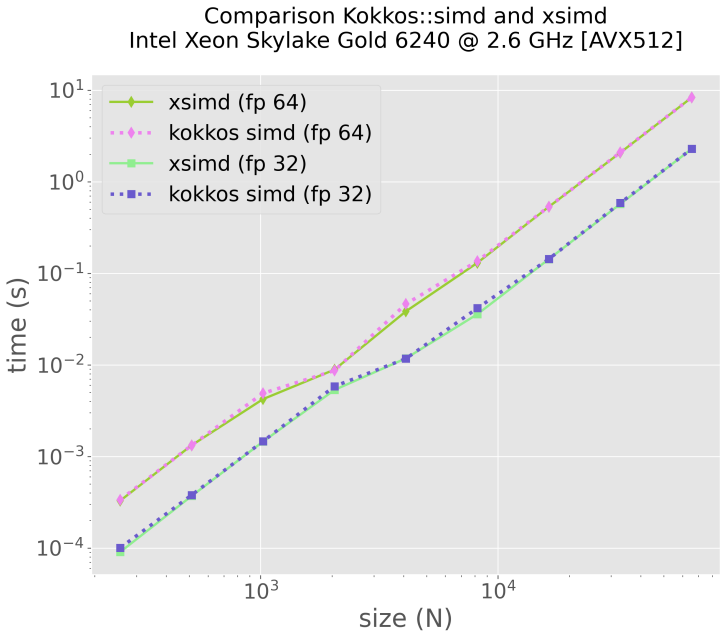

Figure 1: Comparison between xsimd and Kokkos SIMD (Intel Xeon Skylake Gold 6240)

1.5.2 Scalability (with OpenMP threads as a backend)

```
mkdir -p img
python3 scripts/plot-results.py --data-files \
results/bench-kokkos-flat-parallel-for-openmp-time-vs-num-thre
↪  ads-f32-bora012.plafrim.cluster.txt
↪  \
```

```
results/bench-kokkos-flat-parallel-reduce-openmp-time-vs-num-t
↪  hreads-f32-bora012.plafrim.cluster.txt
↪  \
results/bench-kokkos-nested-parallel-openmp-time-vs-num-thread
↪  s-f32-bora012.plafrim.cluster.txt
↪  \
results/bench-kokkos-hierarchical-openmp-time-vs-num-threads-f
↪  32-bora012.plafrim.cluster.txt
↪  \
--colors "coral" "deepskyblue" "forestgreen" "darkgoldenrod" \
--labels "Flat version (parallel_for)" "Flat version
↪  (parallel_reduce)" "Nested version" "Hierarchical version"
↪  \
--linewidths 1 1 1 1 \
--marker d s v o \
--linestyles "-" "-" "-" "-" \
--title $'Scalability with Kokkos backend = OpenMP
↪  (N=65,536)\n(2x18 core Intel Xeon Skylake Gold 6240 @ 2.6
↪  GHz)' \
--xlabel "Number of threads" \
--ylabel "Time (s)" \
--xscale linear \
--yscale log \
--x-data 0 \
--y-data 1 \
--plot-file "img/plot-openmp-time-vs-num-threads.png"
```

1.5.3 Scalability (with C++ threads as a backend)

```
mkdir -p img
python3 scripts/plot-results.py --data-files \
results/bench-kokkos-flat-parallel-for-threads-time-vs-num-thr
↪  eads-f32-bora013.plafrim.cluster.txt
↪  \
results/bench-kokkos-flat-parallel-reduce-threads-time-vs-num-
↪  threads-f32-bora013.plafrim.cluster.txt
↪  \
results/bench-kokkos-nested-parallel-threads-time-vs-num-threa
↪  ds-f32-bora013.plafrim.cluster.txt
↪  \
```

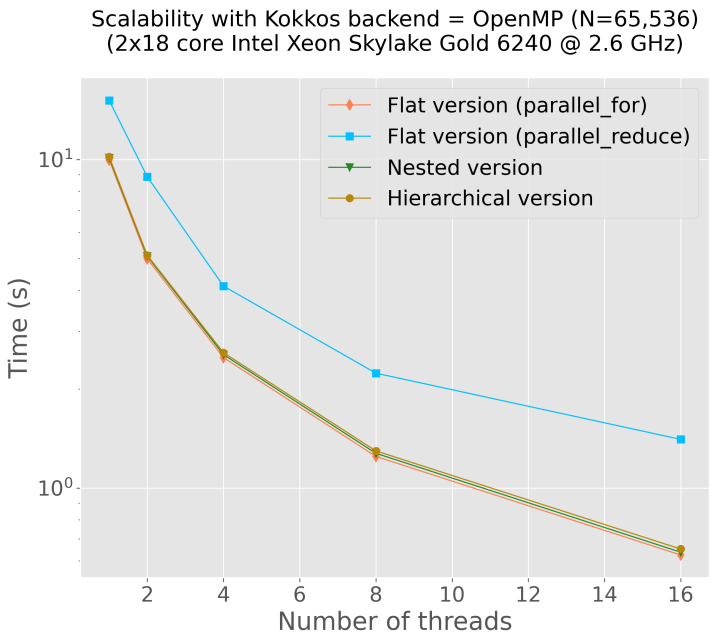

Figure 2: Strong scalability results with OpenMP (Intel Xeon Skylake Gold 6240)

```
results/bench-kokkos-hierarchical-threads-time-vs-num-threads-
↪  f32-bora013.plafrim.cluster.txt
↪  \
--colors "coral" "deepskyblue" "forestgreen" "darkgoldenrod" \
--labels "Flat version (parallel_for)" "Flat version
↪  (parallel_reduce)" "Nested version" "Hierarchical version"
↪  \
--linewidths 1 1 1 1 \
--marker d s v o \
--linestyles "-" "-" "-" "-" \
--title $'Scalability with Kokkos backend = Threads
↪  (N=65,536)\n(2x18 core Intel Xeon Skylake Gold 6240 @ 2.6
↪  GHz)' \
--xlabel "Number of threads" \
--ylabel "Time (s)" \
--xscale linear \
--yscale log \
--x-data 0 \
--y-data 1 \
--plot-file "img/plot-threads-time-vs-num-threads.png"
```

1.5.4 Comparison of the strategies on the GPU (NVIDIA)

```
mkdir -p img
python3 scripts/plot-results.py --data-files \
results/bench-kokkos-flat-parallel-for-cuda-f32-sirocco22.plaf
↪  rim.cluster-experiment-a100.txt
↪  \
results/bench-kokkos-flat-parallel-reduce-cuda-f32-sirocco22.p
↪  lafrim.cluster-experiment-a100.txt
↪  \
results/bench-kokkos-nested-parallel-cuda-f32-sirocco22.plafri
↪  m.cluster-experiment-a100.txt
↪  \
results/bench-kokkos-hierarchical-cuda-f32-sirocco22.plafrim.c
↪  luster-experiment-a100.txt
↪  \
results/bench-cuda-shared-f32-sirocco22.plafrim.cluster-experi
↪  ment-a100.txt
↪  \
```

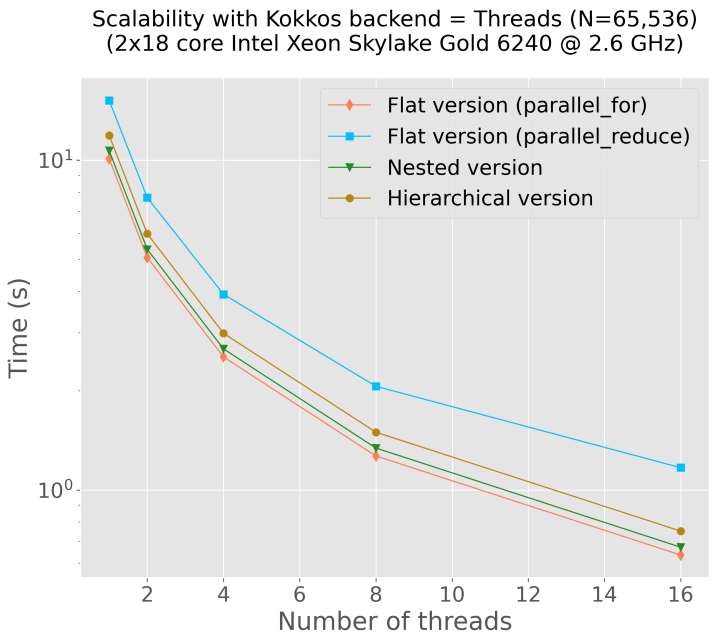

Figure 3: Strong scalability results with C++ threads (Intel Xeon Skylake Gold 6240)

```
--colors "coral" "deepskyblue" "seagreen" "cyan" "red" \
--labels "Flat version (parallel_for)" "Flat version
↪  (parallel_reduce)" "Nested version" "Hierarchical version"
↪  "CUDA (reference)"\
--linewidths 2 2 2 2 1 \
--marker d s o v s\
--linestyles "-" "-" "-" "-" ":" \
--title $'Comparison of the different approaches (GPU: NVIDIA
↪  A100) [f32]' \
--xlabel "size (N)" \
--ylabel "time (s)" \
--xscale log \
--yscale log \
--plot-file "img/plot-kokkos-comparison-cuda-f32-time.png"
```

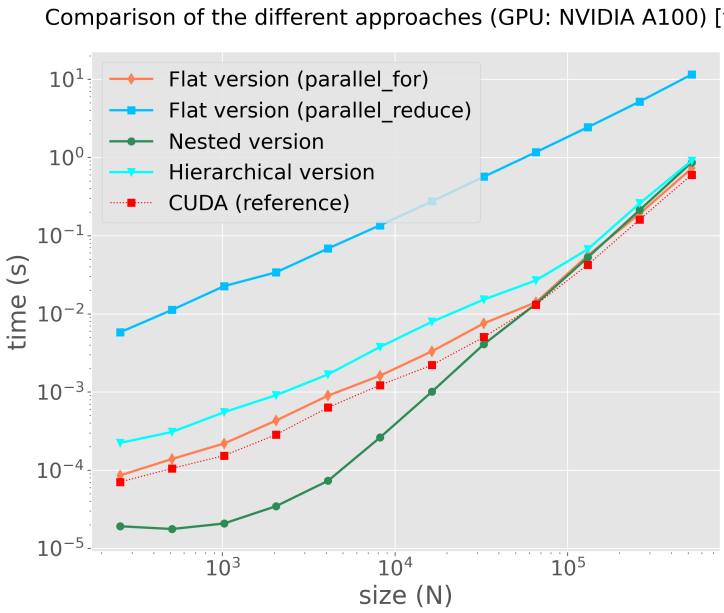

Figure 4: Comparison on GPU (NVIDIA A100)

1.5.5 Observations

In Figure fig. 1, we can notice that both xsimd and Kokkos SIMD achieved comparable performance results on this specific kernel. The final comparison in Figure fig. 4 indicates that the different implementations yielded results similar to those of the CUDA version. However, with the exception of the version that uses the `Kokkos::parallel_reduce` instruction, all other versions delivered similar performance outcomes. Given the use of the **scratch pad memory**, we might have expected significantly better results from the Kokkos-based **hierarchical versions** compared to the flat versions. This discrepancy suggests a need for further investigation, which is currently ongoing. Additionally, it would be valuable to repeat these experiments using a more complex interaction function resulting in a higher computational workload, which is also a work in progress.