



F. Wermelinger
Office: Pierce Hall 211

Homework 5

Peak performance challenge and ISPC

Issued: April 18, 2023 Due: April 30, 2023 11:59pm

Note this is a 2 week exercise. *Do not procrastinate the work.*

Please keep in mind that when the cluster is under high load with many queued jobs, your submitted job(s) will not execute right away. Depending on the cluster load, your job may execute only in a couple of hours (resources are limited). *Make sure you plan ahead.*

IMPORTANT:

- The total duration for this homework is *12 days* (plus at most 2 late days if admissible). Due to Harvard grading deadlines, no other extension is possible beyond this deadline.
- Problem 2 is *optional*. Solving this problem can contribute at most an additional 1 % to your final grade, where the contribution is scaled by the ratio of achieved points relative to the total points of the problem.
- You must upload *individual* submission zip archives to Gradescope due to the optional nature of Problem 2. If you solve Problem 1 only, please upload your submission under “Homework 5 Problem 1” on Gradescope. If you solve both problems, please upload *the same zip archive twice*, once under “Homework 5 Problem 1” and once under “Homework 5 Problem 2 (OPTIONAL)” on Gradescope. **Grading the optional Problem 2 can *only* be considered if your upload exists in the “Homework 5 Problem 2 (OPTIONAL)” assignment on Gradescope.**

Submission instructions: please see the homework submission section in the syllabus.^a

^a<https://harvard-iacs.github.io/2023-CS205/pages/syllabus.html#homework-submission>

Problem 1: Intel Broadwell Peak Performance (60 points)

This problem is a peak performance challenge. Write a small code to reach the highest possible performance on *one Broadwell compute node* on the academic cluster. The objective of this exercise is to reach as much performance as possible with your current knowledge.

Complete the files `kernel.cpp`, `main.cpp` and possibly the `Makefile` to measure the peak *single precision* floating point performance targeting one of the Broadwell compute nodes on the cluster (aim for the 32 core nodes). The code may not compute anything meaningful, your aim is to generate enough floating-point arithmetic instructions in the final executable to saturate the execution units on the hardware. The steps are the following:

1. Compute the roofline ceilings for one Broadwell node on the cluster and draw the hardware ceilings in the roofline plot.
2. Write a small code with an operational intensity in the compute bound region and measure the performance you achieve with your code. Draw *one horizontal line* in the roofline plot from the previous item to indicate your best performance result. Document your approach and explain what are the issues in case you are not reaching close to the compute peak of the node.
3. Report the percentage of the nominal peak performance you achieve.

Rules

- You must use one Broadwell node to collect your measurements. You can either launch a SLURM job or checkout an interactive node. Please do not allocate a full 32 core node for interactive jobs, you can easily develop this problem on a single core and use a proper job script when you test on 32 cores. Use at least the following directives in your SLURM job script:

```
1 #SBATCH --nodes=1
2 #SBATCH --ntasks-per-node=1
3 #SBATCH --cpus-per-task=32
```

You must document how you obtained your results and they *must be reproducible*. Include a shell script in your submission that allows to reproduce your results for example.

- You must use the GCC 12.1.0 compiler to generate your code

```
$ module load gcc/12.1.0-fasrc01
```

- You must use the provided `get_wtime()` timer to collect your time measurements as implemented in the file `main.cpp`.
- You cannot use more software threads than there are *physical cores*. Example: if the target architecture has 16 cores, you can use at most 16 threads. Document the number of threads you have used for the performance result that you report.

- The reference frequency for AVX2 instructions used by the Intel E5-2683v4 processor is 2.5 GHz. Make sure you use this frequency when you compute the nominal peak performance of the architecture. The roofline plot and your reported performance should be relative to this frequency.
- Changing the function signature of `kernel()` in the `main.cpp` and `kernel.cpp` files is allowed, see the TODO's as well. Allocating memory in the main function and passing pointer(s) into your kernel implementation is also allowed (other function arguments are also OK). You may adjust the code in the main function to deal with values that your kernel implementation may return.
- You cannot use ISPC for this problem.

Write your compute kernel in the file `kernel.cpp`. In the `main.cpp` file you must complete the macros `KERNEL_FLOPS` and `ARCHITECTURE_PEAK` which are the total number of floating-point operations in your kernel implementation and the peak floating-point performance of your target architecture in Gflop/s. Your derivation of the number of `KERNEL_FLOPS` must be well documented in your report and in the source code. Writing down just a number without justification is not sufficient, see the file `main.cpp` for more details. Pay attention to compute the total number of flops in your kernel implementation correctly. We will provide an anonymous ranking of the submitted results. A part of the grade for this problem will depend on the achieved fraction of the peak. Run

```
$ grep -n TODO *
```

inside the `code/p1` directory for a list of the open TODO hints.

Hint: Make sure that your code uses SIMD (256-bit AVX2) instructions.

Hint: Use all cores on the compute node. If the thread imbalance output of the main executable is larger than about 6% it means your measurement is inaccurate due to over subscription of threads to physical cores.

Hint: The implementation in `main.cpp` of the handout already provides an OpenMP framework for you, which will launch the `kernel()` function inside a parallel region. You may use `#pragma omp` for loops in your kernel if it makes sense for what you plan to implement. Note that you may not need to compute anything meaningful. Be careful with nested parallelism.

Hint: Compile your code with optimizations such as `-O3` and `-ffast-math` if you use math functions and possibly other flags. See the Makefile.

Problem 2: N-Body Force Computation with ISPC (optional, 40 points)

In molecular dynamics (MD) simulations the interaction force between atoms is often modeled using the Lennard-Jones (LJ) potential. Computing interaction forces is necessary to advance the system in time, since the underlying equations of motion are governed by Newton's law. Computing the gradient of the LJ potential yields a force vector field given by the relation

$$F_{LJ}(\mathbf{r}) = -\frac{12\varepsilon}{r^2} \left[\left(\frac{r_m}{r} \right)^{12} - \left(\frac{r_m}{r} \right)^6 \right] \mathbf{r}, \quad (1)$$

where $\mathbf{r} = \mathbf{x}_j - \mathbf{x}_i$ is the distance vector between atom i and j . The length of the distance vector is denoted by $r = |\mathbf{r}|$ and corresponds to the Euclidean distance. The constants ε and r_m describe the depth of the potential well and the location of the potential minimum, respectively. Computing an update of all particles in the system is of $O(N^2)$ complexity.¹ Because updating particles is expensive, we aim at optimizing the force computation using the SPMD programming model provided by the Intel ISPC compiler. *We are only concerned with single threaded code in this exercise.*

Work in directory: `./code/p2`

a) 24 points

Write a scalar baseline kernel that implements equation 1 for the particle-particle interactions. The `integrate()` function in the application code `main.cpp` will call the force kernel for each particle and expects it to return the components of the force vector in the arguments `Fx`, `Fy` and `Fz`. The constants ε and r_m are provided in the `auxiliary/constants.h` header. You can access them by typing `EPS` and `RM` in your code, respectively.

Write your code in double precision and focus on an optimal implementation by avoiding or minimizing expensive mathematical functions (including division, square root and others), that is, *you are not allowed* to use functions in the `math.h` or `cmath` headers. Try to avoid `if`-branches in loop-bodies if possible. You can compile this scalar version of the code with `make main_scalar` and run it with `./main_scalar`. You can compile your code with debug flags enabled by using `make debug=true <target>`.

Write your code in the file `LJ_force.cpp`.

b) 16 points

Exploit data-level parallelism (DLP) by vectorizing the force kernel with ISPC. Your optimized kernel should run on platforms that support AVX2 instructions, that is, you must use an Intel Broadwell node for AVX2

```
$ salloc -N1 -c32 -t <time you need>
```

To compile your ISPC code, you must complete the recipe in the provided Makefile. Write this recipe for an x86-64 target with AVX2 extensions. You can compile the code with `make` and run it with `./main`.

¹There are methods to reduce this complexity. We are not concerned about these techniques for this problem.

Write your code in the file `LJ_force.ispc` and implement the algorithm in double precision. Report the following for the AVX2 force kernel:

1. The speedup you expect
2. The speedup you achieve
3. If your result deviates from your expectation, explain the reason for the deviation.

Hint: The ISPC compiler is installed on the academic cluster in `$SHARED_DATA/local/bin`.

Hint: Arrays passed to the kernel are in structure of arrays format (SoA).

Hint: ISPC provides a `reduce_add()` function that takes a varying value and returns the sum of that value across all of the active program instances.