

# Hands-on: Polynomials with many floating-point operations

---

- Choose either C/C++ (`cpp`) or Fortran (`fortran`) samples. Both of them are fine, as well.

## C/C++

How to compile and how to execute

### 1. Run a script of creating working space

- See `objst` in `create_project.sh`.
  - It indicates a list of the settings of compile options.
  - Before running `create_project.sh`, please modify `[resource_group]` to the adequate resource group in the `task.sh`.
- Running `create_project.sh`, the executable file (`run.x`) is generated in `Obj_***` directory.
  - This directory is also automatically generated.
  - When you desire to use the own compiler, set `MAKE_DIR` variable in `create_project.sh` from `config` to `config.own`.
- Example:

```
$ cd cpp
$ bash create_project.sh
$ ls
Obj_clang  Obj_clang.512.swp  Obj_clang.inlv2  Obj_clang.novec
```

### 2. Run program

- A job script to execute the program is located in `Obj_***/results`.
- You can run the program either:

```
## To run as a batch job
$ cd Obj_clang/results
$ pjsub task.sh
## Or, to run in an interactive job
$ cd Obj_clang/results
$ bash task.sh
```

- Each of the samples in the Exercises will be completed within 1-2 minutes.
  - For safety, we set the upper limit of job elapsed time as 3 minutes.

## Exercises A

- E1: Check differences between `Makefile.clang` and `Makefile.clang.novec`. Examine the meaning of the compiler options related to the differences.
- E2: Check which of options are related to generating a compiler report in FJ (`*.lst`) and diagnosis message.
- E3: Read the compiler report (`*.lst`) in each of `Obj_***` directories. Also, check the diagnosis message in a log file of the compiler (`make.log`).
- E4: Compare the performance (FLOP/s) in `Obj_clang` to that in `Obj_clang.novec`. You can find the value of FLOP/s in the standard output of the executable file.
- E5: Check the behaviors when setting the compiler options related to optimization on instruction scheduling.
  - Repeat the above exercises, when using `Makefile.clang.inlv2` and `Makefile.clang.512.swp`.
  - Compare the performance (FLOP/s) to that in `Obj_clang`. Which of the compiler options is the best? Furthermore, compare the best result to the peak FLOP/s of a single core in A64FX.

## Exercises B (advanced)

- E6: Test the cases when superword-level parallelism (SLP) vectorization is enable (i.e., `-fslp-vectorize`).
- E7: Test the cases when using Trad mode (e.g., `Makefile.trad.`).
- E8: Measure the cases in `cpp_ExpressionTemplate`. Here, we use a C++ template technique, Expression Template. This technique includes a sort of inlining functions in compiling time.
- E9: In `cpp.fapp`, performance the basic CPUPA analysis using `fapp`. How does the data of cycle accounting change varying compiler options?

## Note

- We summarize the main compiler options of each Makefile.

```
Makefile.clang      #Clang mode. Auto-vectorization with scalable vector length
is allowed.
Makefile.clang.512  #Clang mode. Auto-vectorization with 512-bit vector length
is allowed.
Makefile.clang.512.swp #Clang mode. Auto-vectorization with 512-bit vector length
and software pipelining are allowed.
Makefile.clang.inlv2 #Clang mode. Auto-vectorization with scalable vector length
and interleaving with count=2 are allowed.
Makefile.clang.novec #Clang mode, without vectorization.
Makefile.trad       #Trad mode. Auto-vectorization (w/ 512-bit vector length)
is allowed, but software pipelining is not allowed.
Makefile.trad.nosimd #Trad mode, without vectorization.
Makefile.trad.swp    #Trad mode. Auto-vectorization (w/ 512-bit vector length)
and software pipelining are allowed.
```

## Fortran

### How to compile and how to execute

## 1. Run a script of creating working space

- See `objst` in `create_project.sh`.
  - It indicates a list of the settings of compile options.
  - Before running `create_project.sh`, please modify `[resource_group]` to the adequate resource group in the `task.sh`.
- Running `create_project.sh`, the executable file (`run.x`) is generated in `Obj_***` directory.
  - This directory is also automatically generated.
  - When you desire to use the own compiler, set `MAKE_DIR` variable in `create_project.sh` from `config` to `config.own`.
- Example:

```
$ cd fortran
$ bash create_project.sh
$ ls
Obj_nosimd  Obj_simd  Obj_swp
```

## 2. Run program

- A job script to execute the program is located in `Obj_***/results`.
- You can run the program either:

```
## To run as a batch job
$ cd Obj_simd/results
$ pjsub task.sh
## Or, to run in an interactive job
$ cd Obj_simd/results
$ bash task.sh
```

- Each of the samples in the Exercises will be completed within 1-2 minutes.
  - For safety, we set the upper limit of job elapsed time as 3 minutes.

## Exercises A

- E1: Check differences between `Makefile.simd` and `Makefile.nosimd`. Examine the meaning of the compiler options related to the differences.
- E2: Check which of options are related to generating a compiler report in FJ (`*.lst`) and diagnosis message.
- E3: Read the compiler report (`*.lst`) in each of `Obj_***` directories. Also, check the diagnosis message in a log file of the compiler (`make.log`).
- E4: Compare the performance (FLOP/s) in `Obj_simd` to that in `Obj_nosimd`. You can find the value of FLOP/s in the standard output of the executable file.
- E5: Check the behaviors when setting the compiler options related to optimization on instruction scheduling.
  - Repeat the above exercises, when using `Makefile.swp`.

- Compare the performance (FLOP/s) to that in `Obj_simd`. Which of the compiler options is the best? Furthermore, compare the best result to the peak FLOP/s of a single core in A64FX.

## Exercises B (advanced)

- E6: Test the cases when the policy of software pipelining is changed; `-Kswp_policy=large` and `-Kswp_policy=small`, for example.
- E7: Measure the cases in `fortran_DoConcurrent`. Here, we use `do concurrent` statement in Fortran 2008 standard.
- E8: Measure the cases in `fortran_AssumedShapeArray`. Here, we write the kernel in a modern Fortran manner.
- E9: In `fortran.fapp`, performance the basic CPUPA analysis using `fapp`. How does the data of cycle accounting change, varying compiler options?

## Note

- We summarize the main compiler options of each Makefile.

```
Makefile.nosimd  #Auto-vectorization is not allowed.  
Makefile.simd    #Auto-vectorization is allowed.  
Makefile.swp     #Auto-vectorization and software pipelining are allowed.
```