Bach Khoa University Of Technology
Faculty of Computer Science & Engineering

Course: Parallel Processing
Lab # - Xeon Phi & OpenMP

Anh-Tu Ngoc Tran

February 26, 2018

Goal: understand OpenMP, get familiar with programming model of Xeon Phi and know how to use OpenMP to exploit computing power of Intel Xeon Phi.

Result: have basic knowledge of OpenMP. know how to use and program with Xeon Phi to accelerate your application or computing process.

Contents

1 Basic knowledge

1.1 OpenMP

OpenMP (Open Multi-Processing) is a computing-oriented framework for shared-memory programming. There are a lot of OpenMP directives used to target different directions to parallelize code. Here we only need to understand several basic directives to fully use up all threads and vector registers of Intel Xeon Phi.

- **#pragma omp parallel**: OpenMP uses fork-join model to create a parallel region from master thread.
- **#pragma omp for** shares iterations of a loop across a group of threads. Represents a type of "data parallelism".
- **#pragma omp simd** used to enforce vectorization of loops.

1.2 Xeon Phi programming model

Intel Xeon Phi also known as coprocessor is easy to program, in contrast with CUDA which is hard to learn or program. Basically, when you write a program, it can be run on both CPU and Intel Xeon Phi, or partly on CPU, partly on Intel Xeon Phi. There are two programming models for Intel Xeon Phi: native or offload model. Offload applications are launched on the host CPU and later communicates with the coprocessor. Native applications, in contrast, start on the coprocessor directly.

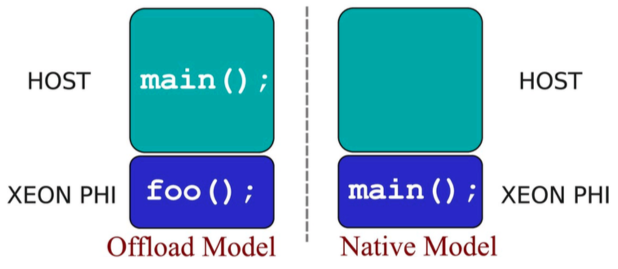

Figure 1: Intel Xeon Phi Programming models

We consider several criteria to decide what programming model we should use: native or offload model. If the problem size is bigger than 16GB, offload model is a good option because KNC only has 16GB memory inside. However, when we use offload model, we have to insert several directives (**#pragma offload target(mic)**) to indicate blocks of code executed on Xeon Phi.

1.3 Vectorization

Intel Xeon Phi has many 512-bit vector registers that can apply mathematical operations to 16 integers or 8 doubles simultaneously. Therefore, maximum potential speedup can be 16 for integer or 8 for double. Intel compiler can automatically do vectorization, or you can explicitly tell compiler to do that by using **#pragma simd**. Furthermore, you can use Intel compiler to generate optimization report to check that the compiler has indeed done vectorization in your executable by using option **-qopt-report**.

1.4 Multithreading

Beside vectorization, multithreading is another source of parallelization. To take advantage of all threads (61 cores x 4 hardware threads) at the same time, OpenMP is used, in this lab, instead of Intel Cilk or

TBB because of simplicity and ease of use. When using multithreading, we have to pay attention to race condition occurring when 2 or more threads writing the same memory address causes unpredictable program behavior. This is resolved by synchronization which only allows one thread access this memory address at a time. However, synchronization is not good for performance and should be avoided if possible.

2 Exercise

With the provided code of matrix multiplication, in this exercise, you will practice programming with Xeon Phi and using OpenMP to parallel code. To specific, you have to insert OpenMP pragmas in the right positions to exploit two main features of Xeon Phi: 512-bit vector registors, and many cores.

Matrix multiplication is an open-ended problem. Until now, researchers are still looking for better answers in parallel to get the result of this computation faster. Some librares that provide good soutions for this problem are Intel MKL, CLBAS and so on.

Follow these steps to finish your exercise. In each step you will do an optimization technique. You have to save your source code in each step to a new file which is different than the provided code:

- Step 1: compile the serial code with Intel compiler and run your executable with command **micnativeloadex**, and record its time.
- Step 2: vectorize your code, generate optimization report and read it to make sure that your code vectorized correctly (**-qopt-report=5**). Then run the executable, record its time, read the report and write to text file the estimated potential speedup of the loop you vectorize.
- Step 3: using **#pragma omp parallel for** to use up all threads on Xeon Phi. Find a correct position to insert this pragma and avoid any syncronization or race condition.

3 Submission

Files to submit (6 files):

- New source file (after inserting pragma) and the executable of step 2
- New source file (after inserting pragma) and the executable of step 3
- Text file (any file word or .txt): note down all the commands you use in shell to compile your code and finish this exercise. Also read the optimization report, write to text file the estimated potential speedup of your code and the time to execute each the executable in each step
- Optimization report generataion by compiler

You have to compress your work in zip format and email to this address - **51304672@hcmut.edu.vn** by **26/3/2018**.

Zip file format: [Class code][Lab2][Student ID].zip
For example: [L01][Lab2][51304672].zip

Otherwise, you will get zero for this exercise. There is no exception for students submitting late or submitting with wrong format. Any problem, please contact **51304672@hcmut.edu.vn** for an answer

4 Reference

For reference you can read the following documents:

- Xeon Phi Programming: `https://colfaxresearch.com/how-series/`
- OpenMP (1): `https://computing.llnl.gov/tutorials/openMP/`
- OpenMP (2): `https://bisqwit.iki.fi/story/howto/openmp/#LoopConstructFor`