Hardware-Software Integrated Systems (HSIS)

Coursework - 2021/22 Academic Year

Module Title: Compilers for Embedded Systems

Module Leader: Dr. Vasilios Kelefouras

DEADLINE FOR SUBMISSION: 20th of February at 23.59

Overview

This piece of coursework consists of two parts.

1. Parallelize and vectorize a software application by using OpenMP (30%)
2. Reduce the execution time of an image processing application (70%)

Part 1: Parallelize and vectorize a software application using OpenMP application programming interface

Download the '*Helmholtz.c*' file from github. This is a program that solves an advanced mathematical problem (discretized Helmholtz equation). Your task is to parallelize the application using OpenMP. You can perform this task either in Linux or in Visual Studio 2019. The marking criteria are as follows.

Marks	**0-4**	**5-9**	**10-19**	**20-30**
Marking Criteria	The student has not used the OpenMP annotations appropriately. The student has not used the OpenMP annotations to all the loop kernels that can be parallelized*.	The student has used the OpenMP annotations appropriately, but just for a few loop kernels. He/she has not used the OpenMP annotations to all the loop kernels that can be parallelized*.	The student has used the OpenMP annotations appropriately to all the loop kernels that can be parallelized*. However, the code delivered includes either multi-threaded code only or vectorized code only, not both.	The student has used the OpenMP annotations appropriately for all the loop kernels that can be parallelized*. The code delivered contains both multi-threaded and vectorized code.

* *If a loop kernel cannot be parallelized or vectorized in its current form, then you do NOT have to take actions against this problem, e.g., the loop kernel in line 178 cannot be vectorized by using OpenMP (in its current form).*

Extra information for Visual Studio users only: As it is explained in the notes of the OpenMP session, Visual Studio (VS) provides limited support for vectorization by using OpeMP. VS support only the *'#pragma omp simd'* clause and not the 'reduction', 'aligned' and 'omp for simd' clauses. The last will give an error, while the other two a warning. Therefore, it is recommended to use Linux. **However, you are allowed to work in VS if you want without losing any marks**.
If you still want to use VS, just follow the instructions below:

- Regarding the '*omp for simd'* clause, it gives an error; if you want to use it, then use the 'omp for' clause instead and put the following comment just after '*omp for simd not supported'*.
- For the *simd reduction* and *simd aligned* clauses, you will get a warning during compilation, e.g., warning C4849 OpenMP 'reduction' clause ignored in 'simd' directive. This means that this clause is not effective and the compiler ignores it. For these two clauses, you can either include them and ignore the warning or you can include them in comments.

Part 2: Reduce the execution time of an image processing application

Drawing upon the optimization techniques that you have learned in this module, you will speed up an image processing application. You can use either Linux or Windows/Mac (Visual Studio). The source code is found on GitHub. In canny.c/canny.cpp file you will find two loop kernels; these are the Gaussian Blur and Sobel. You will optimize the **Sobel loop** kernel only. Please note that there is no single solution.

The optimization includes vectorization using x86-64 SSE/AVX intrinsics, parallelization using OpenMP and register blocking. All the C/C++ Intel intrinsics are provided in the following link: https://software.intel.com/sites/landingpage/IntrinsicsGuide/# . For those who their PCs are old and do not support AVX technology, they can use SSE intrinsics.

The marking criteria are as follows:

Question marks	**0 marks**	**0-9 Marks**	**10-29 marks**	**30-50 marks**	**51-70 marks**
Question.1 marking criteria	The output image is not correct.	The student has not provided appropriate vectorised code using SSE/AVX intrinsics. The output image is correct but parts of the code that can be executed in parallel are not fully vectorised. The student has vectorised the code using OpenMP.	The output image is correct and all the parallel parts are fully vectorised using intrinsics. However, the implementation contains one of the following: A. Bad practice, e.g., exceed the array bounds, B. register blocking and parallelization are not applied.	The student has provided appropriate and efficient vectorised code. The output image is correct, there is no bad practice and all the parallel parts of the code have been fully vectorised. Arrays do **not** exceed their bounds. Register blocking and parallelization are applied. The implementation does **not** contain high latency/throughput instructions like hadd.	**The student has provided an outstanding implementation further reducing the execution time**. This means that more than one output pixels are computed in each iteration and therefore many instructions are saved (both load and arithmetical). Register blocking and parallelization are applied.

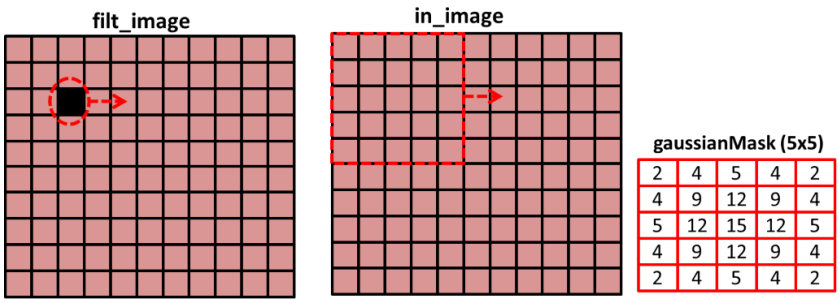

Fig.1 Visual representation of the Gaussian Blur algorithm. The representation of Sobel is similar

Tips:

1. *Try to understand how the algorithm works.*
2. *The mask elements (GxMask, GyMask) contain constant values.*
3. *Before you apply vectorization, fully unroll the two innermost loops.*
4. *There are many different ways to implement this routine using SSE/AVX intrinsics and each solution includes different intrinsics. However, a valid solution exists using the instructions hereafter.*

- *If using AVX intrinsics:*
  - *_mm256_loadu_si256()*
  - *_mm256_maddubs_epi16()*
  - *_mm256_add_epi16()*
  - *_mm256_hadd_epi16()*
  - *_mm256_extract_epi16()*
  - *_mm256_set_epi8()*
- *If using SSE instrinsics:*
  - *_mm_loadu_si128()*
  - *_mm_maddubs_epi16()*
  - *_mm_add_epi16()*
  - *_mm_hadd_epi16()*
  - *_mm_extract_epi16()*
  - *_mm_set_epi8()*

5. *Make sure that the load instructions do not exceed the array bounds. Remember that the 'r0 = _mm256_loadu_si256((__m256i *) & A[i][j])' instruction reads 256bits of data starting from A[i][j], or equivalently 32 char elements.*
   The application of register blocking to vectorized code is the same as applying it to non-vectorized code. The only difference is that instead of using 32bit registers, you are using 256bit registers.

Submission Details

The submission will be done via email to v.kelefouras@plymouth.ac.uk . You will send just the source and header files (do not send images or visual studio files). **Note that if you submit your coursework after the deadline you mark will be capped to 6/10.**