Project 4

CMU 18-640: Foundations of Computer Architecture
Fall 2015

Out: November 17, 2015
Due: 11:59AM EST, December 2, 2015

The goal of this project is to use and analyze the Intel Compute Stick with different optimizations for computationally expensive tasks like matrix multiplications and image processing using *OpenCV*. To start, first download the zip file from here. Unzip to see the contents; they are as follows:

- `mmm` folder: Different versions of code for matrix multiplication along with Makefile.
- `edge_detect` folder: Different versions of code for edge detection along with Makefile and test images.
- `mingw_setup`: Software development environment for Windows (has binary utilities for GNU GCC, Makefile etc.)
- `mingw_packages.txt`: Complete list of packages (some extraneous) for MinGW
- `OpenCL Runtime`: Runtime binary for OpenCL
- `opencv.zip`: Pre-built libraries for OpenCV using MinGW
- `Tight VNC`: To remote desktop into the compute stick
- *win32_153339.exe*: Intel's driver for OpenCL

Step 1: On the new Windows set the username as `cmu_18640` and password as `root`.

Step 2: Install all binaries which includes `mingw_setup`, `OpenCL Runtime`, and *win32_15339.exe*. For `mingw_setup`, select at least the basic packages marked as green in the screenshot `mingw_setup.png`. Also include the pthread library as shown in the screenshot `mingw_pthreads.png`. A complete list of packages can be found in `mingw_packages.txt`.

Step 3: Another library needs to be downloaded and installed for collecting CPU data. Go to Intel's site and download the free version for Windows and only install Intel's VTune Amplifier with Energy Profiler as shown in the screenshot `intel_system_studio_install.png`.

Step 4: Append `MinGW`'s bin folder path to the system path (refer to `mingw_path.png` screenshot).

Part 1 Matrix Multiplication (40 pts)

Matrix Matrix Multiply (MMM) is a well understood problem in high performance computing. Skim Wikipedia page for a review of the basics of MMM.

A naive implementation of MMM incorporates triple loops:

```
#define N 1024// matrix width
void mmmReference(float *A, float *B, float *C) {
  int i, j, k;
  for(j=0; j<N; j++) {
    for(i=0; i<N; i++) {
      for(k=0; k<N; k++) {
      C[i*N+j]=( C[i*N+j]+(A[i*N+k]*B[k*N+j]));
      }
    }
  }
}
```

The code snippet above is quite readable, but its performance suffers when the size of matrices (i.e. N) becomes large enough. The reason is that the capacity of data cache is not big enough to hold the matrices.

A usual technique to optimize the above implementation is via matrix blocking:

```
#define N 1024// matrix width
#define NB 32 //  block width. N and NB must be evenly divisible.
// blocked MMM building block.
void mmm_blocked_building_block(float *A, float *B, float *C) {
  int i, j, k;
  for(j=0; j<NB; j++) {
    for(i=0; i<NB; i++) {
      for(k=0; k<NB; k++) {
       C[i*N+j]=( C[i*N+j]+(A[i*N+k]*B[k*N+j]));
      }
    }
  }
}

// blocked MMM
```

```
void mmm_blocked(float *A, float *B, float *C) {
  int j, i, k;
  for(j=0; j<N; j+=NB) {
    for (i=0; i<N; i+=NB) {
      for (k=0; k<N; k+=NB) {
       mmm_blocked_building_block(&(A[i*N+k]), &(B[k*N+j]), &(C[i*N+j]));
      }
    }
  }
}
```

With proper blocking size (i.e. NB), significant speedup is possible because each small block can be possibly (if no conflicts in cache blocks) placed in data cache while needed.

Another usual optimization is to transpose matrix into column major if this access pattern exists. In our implementation, we transpose matrix B and matrix C for blocked MMM (which is not shown in the above code snippets).

Taking the mmm_blocked() as performance baseline, further speedup can be enabled by leveraging Thread-Level Parallelism and Data-Level Parallelism. The CPU of the Intel Compute Stick incorporates 4 Atom cores, where different threads can execute simultaneously. Besides, each Atom core is featured with SSE4, a Single Instruction Multiple Data (SIMD) implementation, which allows vector processing, aka Data-Level Parallelism.

We already provide you four different implementations wrapped by four different functions:

	Scalar	Vector (SIMD)
Single thread	mmm_blocked()	mmm_blocked_simd()
Multithread	mmm_block_pthread()	mmm_blocked_simd_pthread()

Go to the `mmm` folder. Inside you will see a file for each of the implementation. Go through the `Makefile` file and then execute `mingw32-make` to generate executables.

After compiling, you get 4+1 executables. They are:

- mmm_single_thread_scalar.exe
- mmm_single_thread_simd.exe
- mmm_multi_thread_scalar.exe
- mmm_multi_thread_simd.exe
- *mmm_all_in_one.exe

Among the 5 executables, mmm_all_in_one.exe includes all four implementations and validates their functional correctness with respect to the referenced triple loop implementation. The purpose of this executable is merely to convince you the correctness of these implementations.

Use Intel VTune to collect the runtime characteristics of the 4 executables: mmm_single_thread_scalar, mmm_single_thread_simd, mmm_multi_thread_scalar and mmm_multi_thread_simd.

Collect data for analysis using Intel VTune Amplifier 2015 (**run as administrator**). Open VTune Amplifier, create a project, and under `Application` select the executable `*.exe`. Once the project has been created, click the `New Analysis` button (similar to a play button), and then `Start` button.

Report the following:

1. What can we infer from the CPU usage histogram, for different versions of the code? (3 points)
2. Which instruction type is consuming the most number of cycles? What can be the possible solutions to reduce this? (4 points)
3. What can you say about the CPI and CPU frequency ratio for each execution? How does it vary? (4 points)
4. Modern processors execute many more instructions than the program flow needs. This is called "speculative execution. Are all instructions retired? Give appropriate reason with readings from VTune analysis. (5 points)
5. Table: (24 points)

	mmm_single_thread_scalar	mmm_single_thread_simd	mmm_multi_thread_scalar	mmm_multi_thread_simd
Elapsed time*				
CPU time*				
Effective time*				
CPU Frequency ratio				
CPI				
Context Switch Time				
Instruction type consuming max. time				
Branch misprediction rate				
Cache Hits				
Cache Misses				

Part 2 Edge Detection (45 pts)

In this part we will utilize standard OpenCV libraries for edge and line detection, and analyze multiple implementations for performance using VTune.

Step 1: Extract `opencv.zip`; add `opencv` bin path to the system path variable. For example, if the archive is extracted to the `Desktop` folder then the path to be added would look like:

- `C:/Users/cmu/Desktop/opencv/mingw_build_st_nsse/install/x86/mingw/bin`

Step 2: Edit the `Makefile`. Specifically edit the paths to libraries and headers based on the location of the `opencv` folder extracted in the last step.

Step 3a: Open a command prompt `cmd`, and go to the `edge_detect` folder unzipped from the original archive and compile the single threaded no SSE (`st_nsse`) using `mingw32-make st_nsse`; and run the executable `st_nsse.exe`. The `edge_detect/out` folder should contain the resulting images. Note that we already edited the system path to point to `st_nsse` binaries in Step 1.

Step 4a: Collect data for analysis using Intel VTune Amplifier 2015 (**run as administrator**). Open VTune Amplifier, create a project, and under `Application` select the executable `st_nsse.exe`. Once the project has been created, click the `New Analysis` button (similar to a play button), and then `Start` button. Although, when using Vtune to collect data, make sure to add the OpenCV bin path as `Working directory`. For an example see `vtune_working_dir.png` for `mt_nsse.exe`.

Similarly, one can compile and run the single thread with SSE (3b); multiple threads without (3c) and with SSE (3d); and finally the `OpenCL` with multiple threads (3e). For **each compilation and run**, first add the correct `opencv` bin path to the system path variable as done in Step 1, and then open a new command prompt terminal. If you wish to only create the executable and not run it from the command prompt, then you may skip editing the system path variable. Collect data for each executable - (4b) `st_sse`, (4c) `mt_nsse`, (4d) `mt_sse`, and (4e) `mt_sse_ocl`. Source codes for with and without SSE are similar.

Report the following:

1. Describe the impact of context switch time consumptions and wait times? (3 points)
2. How much does branch misprediction affect the execution times of the program? (4 points)
3. Memory: What are the aspects with respect to memory that can be taken care of in order to improve performance? Report the parameters that are crucial with respect to memory. (4 points)
4. What is Page Walk? How does it influence performance? (4 points)
5. Table: (30 points)

	st_nsse	st_sse	mt_nsse	mt_sse	mt_sse_ocl
Elapsed time*					
CPU time*					
Effective time*					
CPU Frequency ratio					
CPI					
Context Switch Time					
Instruction type consuming max. time					
Branch misprediction rate					
Cache Hits					
Cache Misses					

Part 3 Digit Detect (15 pts)

In the `edge_detect` folder locate the `digit_detect.cpp` source file. Use `mingw32-make st_digit` which will create `st_digit` executable. Use VTune to analyze it.

Report the following:

1. What are front-end and back-end executions in superscalar processors? (4 points)
2. Apart from mentioned, describe at least two other key parameters that can be used from VTune to transform our programs performance? (6 points)
3. Table: (5 points)

	st_digit
Elapsed time*	
CPU time*	
Effective time*	
CPU Frequency ratio	
CPI	
Context Switch Time	
Instruction type consuming max. time	
Branch misprediction rate	
Cache Hits	
Cache Misses	

Part 4 Bonus (10 pts)

Parallelize Digit Detect code to make it run faster. Feel free to try out any approach to parallelize the code. Explain your implementation and analyze the parameters as in part 3.

Part 5 Handin details

You will need to hand-in the Report (in PDF) and code for the bonus part to
`/afs/ece/class/ece640/submission/group<group_no>/project4/`.

*all time measurement with Advanced Hotspot Analysis