PyCompArch: Python-Based Modules for Exploring Computer Architecture Concepts

Workshop on Computer Architecture Education 2015

Dan Connors, Kyle Dunn, Ryan Bueter
Department of Electrical Engineering
University of Colorado Denver
Overview

- Motivation
- IPython Notebook
- PyCompArch Concept Notebooks
  - Multiprocessing: Mandelbrot, Amdahl’s, Speedup, and Efficiency
  - ParallelSchedule: Job Scheduling
  - Synthetic Speedup: Interactive
- PyCompArch Concept Notebooks
  - Dynamic frequency scaling
  - Computer Vision (CV)
  - Code Box
Motivation

- Visualization of concepts of parallelism
  - Explore concepts from learner’s perspective versus single static graph of parameters
- Explore/perform experiments on hands-on devices
  - Raspberry PI
  - Streamline collection, comparison, and visualization
The IPython Notebook is a web-based interactive computational environment. It offers a well-structured code development environment, a framework for observing and recording results of code execution, linking text such as comments and equation generators for mathematics, embedded plots and other rich media formatting options. The cloud coding advantage is that the IPython Notebook Viewer renders the code as a web page and users can read and interact with a remote system without having to install anything on their device. Changes can be rolled back, encouraging experimentation without creating excessive copies of source material.

Simple spectral analysis

An illustration of the Discrete Fourier Transform

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-2\pi kn/N} \quad k = 0, \ldots, N-1 \]

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a dataset using SciPy’s audio file support:

```
In [1]: from scipy.io import wavfile
    rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib’s built-in `specgram` routine:

```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
    ax1.plot(x);
    ax1.set_title('Raw audio signal')
    ax2.specgram(x); ax2.set_title('Spectrogram');
```
Example: Jupyter (www.jupyter.org)

```
In [ ]: %matplotlib inline

import pandas as pd
import numpy as np
import matplotlib

from matplotlib import pyplot

ts = pd.Series(np.random.randn(1000)
    ts = ts.cumsum()

df = pd.DataFrame(np.random.randn(1000)
    columns=['
    df = df.cumsum()
    plt.figure(); df.plot(); plt
```
Python

- IPython Notebooks
  - Interactive lectures
  - Code and see: embed visualization of parallel performance trade-offs
  - Assignment template
  - PyMTL is a hardware modeling framework
  - Pydgin is a framework for rapidly developing instruction-set simulators (ISSs) from a Python-based architecture description language.
IPython

- Computer Architecture Concepts - Python Parallelism
  - Multithreading and multiprocessing support
  - Speedup and efficiency
  - Amdahl’s law
  - Overhead

- Explore Experiments
  - Benchmarking (Example: OpenCV)
  - Collecting, displaying, and comparing results
  - Dynamic frequency scaling on Raspberry Pi
Mandelbrot Calculation : Block Size

➢ Overhead of work assignment

```python
import time
size = 1000

for block in [500, 5000, 50000, 500000]:
    tic = time.time()
    z = create_mandel(-2.2, 0.8, -1.5, 1.5, size, block)
    print(block, time.time() - tic)
```

500  8.53376102448
5000  4.11379504204
50000  3.75533103943
500000  6.21821498871
Parallelism – Python’s Multiprocessing Module

Parallel: 3 steps

Step 1: import

```
In [15]: import multiprocessing
```

Step 2: Create a pool of workers.

```
In [16]: num_cores = multiprocessing.cpu_count()
    pool = multiprocessing.Pool(num_cores)
    print(num_cores)
```

Step 3: Modify the serial map.

```
In [17]: def create_mandel(min_x, max_x, min_y, max_y, size, block_size, pool):
    x_vals = np.linspace(min_x, max_x, num=size)
    y_vals = np.linspace(min_y, max_y, num=size)
    mandelbrot = grid(x_vals, y_vals, block_size)
    results = pool.map(mandel_z, mandelbrot)  # <- Modification
    output = np.concatenate(results)
    return output
```
Speedup and Efficiency

speedup = \frac{\text{sequential time}}{\text{parallel time}}

S = \frac{t_1}{t_n}

Measures processor utilization

\text{efficiency} = \frac{\text{sequential time}}{\text{(number of processors} \cdot \text{parallel time})} = \frac{\text{speedup}}{\text{num of processors}}

\eta = \frac{S}{n}

In [21]:
import pandas as pd
df = pd.DataFrame({'n': cpus, 't': t})

In [22]:
df['speedup'] = df[df['n'] == 1]['t'].values / df['t']
df['eff'] = df['speedup'] / df['n']
Base Graph

![Graph showing linearity with speedup and number of cores. Serial fraction = 0.0%. Overhead fraction = 0.00%.]
Parallelism Exploration

In [6]: `StaticInteract(plot_speedup, s=RangeWidget(0, 10), o=RangeWidget(0, 10))`

Out[6]:

Slider control in IPython figure
Parallel Exploration

➢ Serial fraction

➢ Overhead
Execution Time

![Graph showing the relationship between time (sec) and number of cores. The time decreases as the number of cores increases.](image)
Speedup

- mandel
- ideal
Efficiency
Amdahl Law's Evaluation

Amdahl's Law

In [49]:
parallel_ = 0.9
serial_ = 0.1
total_time = df.ix[0,'t']

np = range(1, 13)
t = [ total_time * (serial_ + parallel_/float(n)) for n in np ]

In [50]:
fig, ax = plt.subplots(figsize=(8,6))
ax.plot(df['n'], df['speedup'].values, label='actual')
ax.plot(np, total_time/t, label='Amdahl')
ax.set_xlabel('number of cores')
ax.set_ylabel('speedup')
ax.legend(loc='upper left')
fig.show()
Amdahl Law’s Evaluation
Single Job Timeline
Parallel Job Scheduling Timeline (8 cores)
Worker Workload Summary
Computer Architecture Education and Raspberry PI

- Center piece of course
Raspberry PI Stats

- $35 is inexpensive...must be a toy?
- “It’s just a slightly underpowered computer without a screen, and anything you can do on it you could do on a laptop”
- True, but also:
  - Students can’t destroy the systems with software.
  - It changes mindsets because the systems are easily accessible.
- What about Arduino?
  - Similar aims, but Raspberry Pi runs Linux- full operating system, is a modern 32-bit ARM processor
OpenCV Algorithm Evaluation on Raspberry Pi

- Evaluate selected set of various OpenCV functions
  - rotate, convolution, sobel, median blur, resize, histogram, erosion, etc
  - Benchmark various image file sizes: 720x480 1280x720 1920x1080 3840x2160
Raspberry Pi- DFS (Dynamic Frequency Scaling)
In [5]:
    def access(x, y, n):
        code = ""
        unsigned long int i;
        for (i=0; i < n; i++) {
            *y = x[i];
            y++;
        }
        ""
        weave.inline(code, ['n', 'x', 'y'])

[6]:
    def strided_access(x, y, n, stride):
        code = ""
        unsigned long int i;
        for (i=0; i < n; i++) {
            *y = x[i*stride];
            y++;
        }
        ""
        weave.inline(code, ['n', 'x', 'y'])
Code Box

➤ Execution time observed

```python
In [26]: num_range, times = function_timing(access)
plt_times(num_range, times)
```

![Graph 1](image1)

```python
In [27]: num_range, times = function_timing(strided_access)
plt_times(num_range, times)
```

![Graph 2](image2)
Summary and Future Plans

- New opportunities for exploring concepts related to computer architecture
  - Theoretical concepts – Amdahl’s law
  - Evaluation of Python multiprocessing module for parallelism
  - Code Box – Evaluation of performance of small code examples
  - Sharing content between developers [github]
- Raspberry Pi support
  - Enhance development of projects and independent learning by having set functions for gathering performance results
  - Dynamic frequency scaling
  - OpenCV evaluation
- Future work: Python-based Numba generates optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool).
  - Numba supports compilation of Python to run on either CPU or GPU hardware.